1999

Accurate visualization of distributed system execution

Dennis Lee Edwards
College of William & Mary - Arts & Sciences

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

ACCURATE VISUALIZATION OF DISTRIBUTED SYSTEM EXECUTION

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

---

by

Dennis L. Edwards

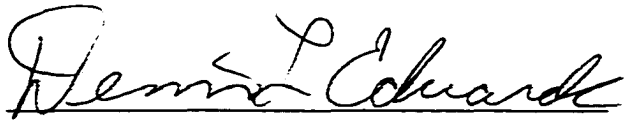
1999

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of

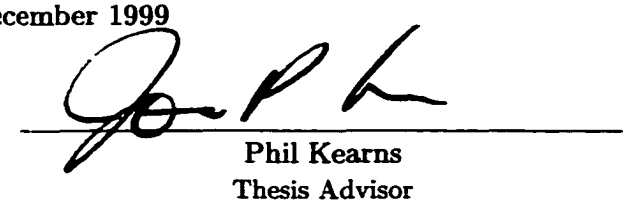
the requirements for the degree of

Doctor of Philosophy

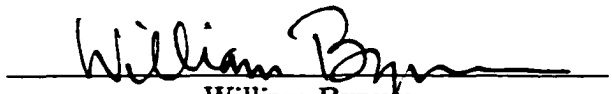


Dennis L. Edwards

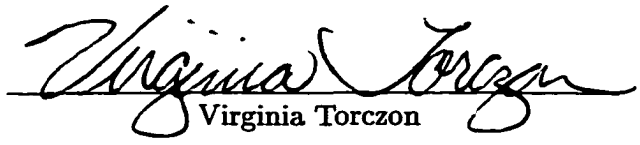
Approved, December 1999



Phil Kearns
Thesis Advisor

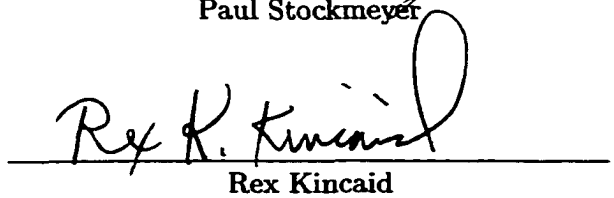


William Bynum



Virginia Torczon

Paul Stockmeyer



Rex Kincaid
Department of Mathematics

To my parents and brother.

Thanks for never giving up.

Table of Contents

List of Figures

ABSTRACT

The concurrent execution of processes in a distributed system makes the interactions between them difficult to understand. A clear image of the execution sequence that occurs as well as possible alternative scenarios is paramount to providing the software engineer the understanding needed to create reliable software.

In this thesis, we examine several representation methods of presenting a pictorial view of the execution of a distributed system and evaluate each in terms of three criteria. First, only those relationships that are created during the execution can and must be presented. Second, the presentation must be such that a reasonable amount of information can be extracted from the picture. Third, the technique must be scalable to an arbitrary large system.

We begin by examining an informative and scalable technique. We expose a simple graph that, if present in the system's execution, prevents accuracy from being achieved in this technique. We prove that the absence of this graph is sufficient to ensure that an accurate representation is possible.

Our own technique is developed next. We first prove that the technique is accurate in that a representation of an interprocess relationship is presented if and only if it actually exists in the execution being displayed. While being theoretically scalable to any number of processes, the practical use of the technique is limited to a small distributed system that is relatively short-lived. In addition, the display is of only moderate use since it quickly becomes too complex for information to be extracted by the software engineer.

We conclude by presenting our second technique that is accurate, scalable, and informative. The technique does not attempt to display the complete set of relationships in a single picture. Instead, an accurate subset is presented and the engineer is allowed to easily alter the selected subset. We show that the presentation is clear and concise. An implemented prototype demonstrates the utility of the technique in providing the software engineer with a tool that can be used to develop software that is less likely to suffer from concurrency related problems.

ACCURATE VISUALIZATION OF DISTRIBUTED SYSTEM EXECUTION

Chapter 1

Introduction

Parallel and distributed programs are difficult to write. Not only are the problems generally larger and more complex than those solved with sequential programs, but the solutions themselves are more complex. The processes that comprise the system operate autonomously, communicating through messages. It is this individuality that gives the system its power and its complexity.

In order to write correct code, the software engineer must understand the interactions among the constituent processes. A visualization technique will convey the interactions quickly to the engineer. However, the technique cannot be arbitrary. It must maintain the following three properties to be considered acceptable.

Property 1.1 *Accurate*

> It is not acceptable to present the software engineer with a representation that indicates the impossibility of an occurrence when that possibility does exist. Likewise, a representation indicating that an occurrence can happen when it is in actuality not possible, is not acceptable.

Property 1.2 *Informative*

> It is not acceptable to present the software engineer with a representation from which valuable information cannot be extracted. Although the needed information may not be immediately obvious, it must be obtainable though moderate cognitive exertion.

Property 1.3 *Scalable*

> It is not acceptable to compose a representation that is adequate for only small systems. The technique must be scalable in, preferably, a linear fashion to an arbitrarily large distributed system.

A technique that meets all three properties will be considered acceptable. We now look at three classifications of research results that have taken positive steps toward these acceptance goals. Although it is not a complete taxonomy of execution visualization systems, these three classifications are adequate for the division of techniques representative of the current literature. Few techniques fit into a single category. Instead, most contain methods representative of each class.

The first classification is for systems that display the concurrent strength of the system as measured in terms of possible concurrency. The objective of these techniques is to provide feedback to the software engineer to facilitate more optimal solutions to the problem at hand. These techniques, categorized as *performance* models, dominate the current literature.

The second classification of visualization techniques is designed with the elimination of logical errors in mind. These techniques are extensions in the distributed domain of

sequential debuggers and offer limited insight into the complexities of the examined system. The category for these techniques is *debuggers*. We have seen several recent entries into this category.

Last we have the *graphical* class. This is a hodgepodge of visualization methods dedicated to the explanation of execution possibilities. Each is designed to foster further understanding of the system. Most of the entries in this category could also be placed in one or both of the previous categories. We now look at some typical examples of each class.

1.1 Performance Tools

A common feature of most performance monitoring and display techniques is their presentation of quantitative information using three standard techniques: Ghantt charts, Kiviat graphs, and time-space diagrams. Ghantt charts and Kiviat graphs are used to display metrics such as the number of I/O references, CPU memory references and event occurrences. Time-space diagrams (discussed in detail later) present message passing information depicting communication patterns. The techniques that display concurrency do so as a relative measure compared to possible speed-up. For example, the Kiviat graph shown in figure 1.1 shows that processes numbered $0, 3, 5$ and 7 are executing at a near optimal level while processes numbered 2 and 4 are performing poorly.

Traceview[30] uses Ghantt charts to present several metrics including concurrency which is a percentage of a theoretic maximum as derived by symbolic execution of the source code. ParaGraph[23] employs all of the listed display techniques and includes facilities for incorporating new techniques as they are developed. The Parade[42], Kanoko[34] and Pavane[38]

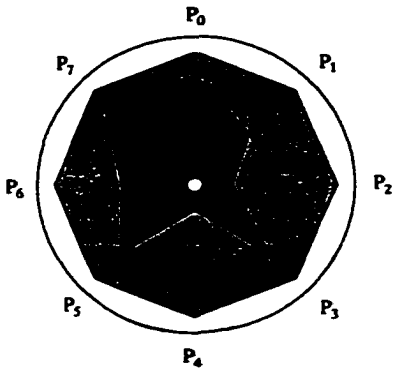


Figure 1.1: An example Kiviat graph

visualization environments create different views of program execution specifically for the application being examined. For example, a system modeling the interaction of molecules may be displayed as a black box with floating spheres. The views created by Parade are built with the Polka[43] animation system that allows the view to be altered as the system parameters change. Kanoko and Pavane differ in creating three dimensional animations where Polka views are two dimensional. Hart, et al[22] use the Pavane system along with two dimensional graphs and charts to present distributed computations. Continuing with the molecular example, the spheres could be animated to coincide with the program's computations.

The visualization component of the AIMS[27] toolkit is composed of multiple "kernels", each with a different display type. The ACTS[40] toolkit is designed to foster the creation of parallel, object-oriented programs by displaying processor and message activity on a time line. It also presents several views of profile information derived from the system execution. The Meander[46] programming environment provides the software engineer a graphical environment to construct parallel programs as a collection of program components.

A visualization system allows the execution of the program to be displayed as interactions among the components.

A time tunnel[37, 33] arranges processor activity in a cylindrical fashion with the point of view placed at one end of the cylinder. Lines along the edge of the cylinder represent the execution of a particular process with message transfer depicted as directed arcs connecting the process lines. This type of display becomes unusable as the number of messages increases even with the stereoscopic displays available with some systems.

An activity matrix is the only graphical display available with MPPE[31] from MasPar. In a system with 1024 processing elements, a 32×32 binary matrix is presented. An "on" bit in element a, b represents processor activity in that processor. Work by Stavely[44] uses derivatives to characterize the concurrency of the system but offers no direct implementation as a display technique.

SMILI[28] represents a method of displaying gathered system or process statistics in a way that possesses a "mnemonic advantage" by relating program function to a human emotional state. As the authors have noted

> ...not much [information] can be extracted if one has to scan 128-point Kiviat graphs or a Ghantt chart displaying processor activity for 128 nodes.

Here each variable of a p-dimensional data set is represented as a feature on a cartoon face with "bad" data being represented as a sad or angry face and "good" data being represented as a happy face. Some of the features that may be used to represent data include the height and length of the brow, the height and width of the eye, nose and mouth, and the overall ovalness of the head.

Figure 1.2 shows faces representing both a "good" execution and a "bad" execution. Suppose that the shape of the mouth is used to characterize concurrency. A grin would indicate optimal execution while a frown would indicate poor performance. Data that falls between the two extremes would be presented as an arc somewhere between a grin an a frown depending on a value relative to optimal.

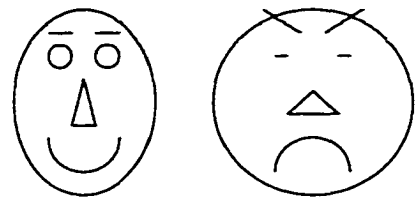


Figure 1.2: A "good" execution and a "bad" execution

Data can be presented in either of two ways. In one case, the data represents a complete execution of the system. This allows different runs to be compared. The second case presents the execution of a single process. Here, process function is compared. In either case, all the data is clustered into a single picture.

Recently, Kanoko[34] has extended the use of faces into three dimensions using the work of Chernoff[9]. While the informative content has not changed, the aesthetic qualities have improved. Also included in the Kanoko display technique are aural indications of processor activity and color-coded simulations designed specifically for the application being displayed.

1.2 Debugging Tools

Some distributed debuggers are simple extensions of sequential debuggers with limited execution control imposed via message passing commands. Ddbx-LPP[13], Blit[7] and

DPDP[47] have replaced the control commands **next**, **step**, etc., with their distributed counterparts. Each allows breakpoints to be inserted into a process and examination of variables and context of stopped processes but provides no visualization other than highlighted source text. The Blit debugger does provide a window interface to its facilities, but the visual extension ends there. Multiple windows are also provided by DPDP with the source code of a single process occupying each window. An arrow indicates the current statement, and relationships between statements are only visible through the constraints placed on execution steps.

Other debugging tools, for example, Xab/Hence[4] and I-Pigs[36], represent the system as a collection of visually connected processes. The Xab/Hence technique colors each process to indicate its current state: not ready, executing, completed, exited, warning or error. I-Pigs not only colors processes to indicate either executing or waiting status but also highlights process connections to indicate message transmission. Concurrency in each case is shown as multiple processes in the executing state. Possible relations other than those derived from the execution order are ignored. A graphical programming environment is used to construct the displays available in PDG[6]. Trace records obtained through a specialized debugging kernel are used in conjunction with the graphical specification of the distributed system being debugged to create a representation used to animate the program. Although this method holds certain promise, errors in the original specification may produce anomalous behavior in the display of an otherwise correctly operating program.

Task graphs form the basis of TraceViewer's[25] display. Function calls written in IBM parallel Fortran are displayed as event occurrences with directed vectors indicating the direction of the call. The technique allows the selection of a "current event" and then

provides facilities to highlight those events that happen before, are concurrent to, and happen after the selected event. Boolean operators are used to combine multiple queries.

1.3 Graphical Tools

We will examine three representatives from the graphical class of visualization tools. The tools describe the relationships between events of the system with varying degrees of accuracy and informative content.

1.3.1 Time-Space Diagram

One of the first methods used to display the execution of a distributed system was the time-space diagram[29] shown in figure 1.3. Each vertical vector represents the execution

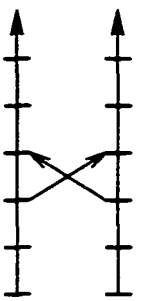


Figure 1.3: A *time-space* diagram

of one constituent process. Hash marks on the process line indicate the execution of an event within that process. The temporal order of events of a process is indicated by the total ordering of events on the vertical time line. Horizontal separation between process lines represents physical spatial separation of processes. Directed arcs connecting events on separate processors identify message transmission and receipt.

This technique displays the transitive reduction of the relationships between events and neither looses nor falsely includes relationships between events. Therefore, property 1.1 is obtained. While it is easy to deduce the relative order of events that are closely connected in the graph, it is difficult to determine the order of remote events. Therefore, property 1.2 is violated. This technique is also difficult to scale to any system larger than a "toy" which violates property 1.3. For these reasons it is not generally considered a viable method of expressing the execution of a generic distributed system but is an alternative for less complex systems.

Time-space diagrams are one visualization technique used by Vampir[18], a trace visualization toolkit designed for use with MPI. The processor lines include a color code to indicate the state of the process at that time. Color codes are assigned to the time lines according to the values of local processor clocks. Messages are then added to the already constructed time lines. It is possible that temporal anomalies are created if process clocks are not closely synchronized. A message could arrive at a time prior to the time of transmission thereby resulting in a backwards arc in the display.

1.3.2 Hasse Diagram

A Hasse diagram[21] is used by Cooper and Marzullo[11] to display the execution possibilities of a distributed system. Each node represents a set of events. The set contains one event from each process. Events in a node are capable of executing given the previous nodes of the graph have occurred. A node containing $(2, 3)$ indicates that the second event from the first process and the third event from the second process are ready to be executed. Directed arcs indicate the occurrence of a single event. Multiple arcs emanating from single

node represent multiple possible event executions. Each path through the diagram defines a total ordering of events consistent with the partial order defined by the system execution.

The Hasse diagram of figure 1.4 represents the same execution shown in figure 1.3. Several observations about this type diagram are immediately obvious.

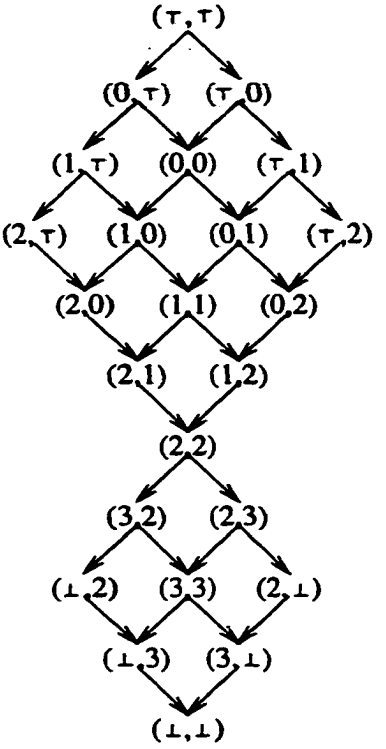


Figure 1.4: A *lattice* of possible event occurrences

- The width of the tree is an indication of the concurrency present in the system. A tree with only a single path from the initial to the final node is representative of a sequential program.

- The nodes of the diagram can be transformed from system events into system states. In this case, each node would represent the state of the system at the point where all

events up to those in the current node have been executed. Arcs are then viewed as state transitions resulting from event execution.

- Synchronization points are identified by the narrowing of the diagram. Referring to the above example, the system must enter state $(2, 2)$ before either process can continue execution.

- In the worst case, the width of the lattice grows exponentially with the number of processes in the system.

We see that all possible executions are depicted, thereby satisfying property 1.1. To determine whether or not selected events are concurrent, the entire lattice must be searched for a state containing the events. However, the last bullet signifies the major weakness associated with lattice representation – the exponential growth as system size increases. The width of the graph is directly proportional to the amount of concurrency in the system. As the size of the system increases, so does the concurrency and, therefore, the graph.

1.3.3 Concurrency Map

The concurrency map[45] uses a single static diagram to display the relationships between all events of the modeled system. Events are placed in a two-dimensional grid where the column indicates the process where the event occurred and the row indicates the relative order of events. In the left column of figure 1.5 the send event is shown to occur before the receive event in that process. Two events can occur simultaneously only if they are displayed at least partially in the same row. Referring again to the figure, the two send events can occur simultaneously since they are both included in the first row.

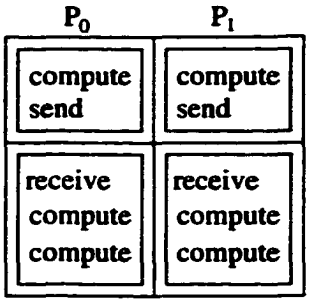


Figure 1.5: A *concurrency map* of event occurrences

Linear scaling is achieved since the inclusion of another process adds a single column to the grid structure. As the figure demonstrates, it is a simple matter to discern the relative order of events in a concurrency map. However, display accuracy is sacrificed to achieve the other two properties. Under some conditions it is not possible to display all relationships in the system. For this reason, we consider this method unacceptable.

1.4 Structure of the Dissertation

Each technique we have examined has had both positive and negative aspects. Most provide valuable information limited to its designed purpose. In chapter 2, we present the basic model of a distributed system and provide a brief discussion of abstractions used to reduce the complexity of the underlying system. This model is then used in chapter 3 to explain the failure of the concurrency map. Chapter 4 will present a general technique based on the information obtained from the concurrency map that is accurate but not optimal according to our required properties. In chapter 5 we will describe an accurate technique that partially fulfills the other properties. Conclusions and future research are detailed in chapter 6.

Chapter 2

The Basic Model

2.1 Distributed Systems

The notion of program execution in a sequential environment is well-defined. A statement is executed at some time τ and at a later time, $\tau + \delta$, the next statement is executed. A process is instantiated at the beginning of execution and remains live until execution has completed. Even in multiprocessor machines where executing programs migrate between processors, execution remains sequentialized. A single locus of control is maintained until the program is terminated.

A value computed by one statement that is needed for execution of another statement is a simple matter in a sequential environment. The value is written to a memory location and later read from the same location. Communication between statements is accomplished via direct memory accesses. We assume that no shared memory exists in the distributed environment.

Distributed applications are designed to profit from the advancements in network technology. Individual machines cooperate to collectively solve a problem that either cannot be solved by a sequential program or that requires an unacceptable amount of time by a single

machine. Each machine is given a program that has been designed to solve a portion of the problem. The machine will instantiate a *process* to execute its program just as in the sequential case. In the remainder of this work we will refer to the executing portion of the program assigned to a single processor as a process.

Definition 2.1 *A distributed system is composed of a set of N processes,* $\{P_0, \ldots, P_{N-1}\}$, *cooperating to solve a problem.*

2.1.1 Communication Paradigms

As the processes execute, they may need to communicate intermediate results to other processes. Each communication datum is encapsulated in a message and sent across a network. The process originating the message is the *sender* and the target process is the *receiver*. We first consider only well-behaved networks that function without loss and in a point-to-point, first-in-first-out manner. That is, if two messages are sent from process P_i to process P_j, both messages are guaranteed to arrive at P_j in the order P_i sent them. No restrictions are placed on either the time a message is in transit or on the receipt order of messages with different senders or receivers.

Several types of message passing appear in the literature. The two most prominent are *asynchronous* and *synchronous*. Messages sent in asynchronous mode are commonly moved from the memory space of the sender to space reserved for the process implementing the network facilities. At that point, the sending process continues to execute without further interruption. The physical transfer of the message from the machine executing the send to the machine executing the receive takes place without the interaction or knowledge of either the sender or receiver. When the message arrives at the recipient machine, the receiving

process can request the message to be placed in its address space. The only execution interruption that is present in this type of communication model is at the receiving end. If 1) the receiver process requests a message from the underlying network facilities, 2) the message either has not yet arrived or has arrived and is not yet ready for delivery, and 3) the receiver is in *blocking* mode, then the receiver will block, suspending execution and wait for the delivery of the message. If the message is ready for delivery or the receiving process is in *non-blocking* mode, no interruption of execution occurs.

Conversely, process blocking is the basis of synchronous communication. When either the sender or the receiver executes a communication statement, execution is blocked until the matching statement at the other process is also executed. Only when both the transmission and receipt of a message can be executed, is the message transferred and execution resumed. When a matching send/receive pair is executed, the processes involved attain a temporary synchronization, or *rendezvous.*

2.1.2 Events

The execution of a single process is modeled as a totally ordered sequence of discrete actions. Each of these actions is assumed to be atomic and instantaneous. In practice, an action is begun, executes for some amount of time, and then completes. Our model uses either the time of initiation or completion as the representative time of the action. The choice is arbitrary but must be consistent. Individual actions are referred to as *events.* For simplicity, we let an event represent the execution of a single statement from the source code of the process.

Definition 2.2 *The execution of a statement of a distributed program is an event v.*

Definition 2.3 *The function $\mathcal{P}(v)$ identifies the process in which the event v was executed.*

The k^{th} event to be executed in process P_i is denoted v_i^k. Although events correspond to the execution of program statements, event order and statement order may be different. Statements are ordered as lines in the source code of a program while events are ordered by their temporal occurrence during the program's execution. Each process is assumed to have an initial event $v_i^\perp$ and a terminal event $v_i^\top$. Let $E_i = \{v_i^\perp, v_i^0, v_i^1, \ldots, v_i^\top\}$ be the set of events executed in process P_i, and let E represent the set of all events in the system, including system initial and system terminal events.

$$E = \bigcup_{i=0}^{N-1} E_i \cup \{v^\perp, v^\top\}$$

Each event is classified as *send*, *receive* or *computation*, and is referenced by the sets S, R, and L respectively. (Computation events, by nature, directly effect only the state of the executing process and are therefore considered "local.") The set of communication events is collectively referred to as $M = S \cup R$. In our context of message-based communication, a send refers to the transmission of an encapsulated message and a receive refers to the receipt of an encapsulated message at the destination process. Given that m represents a unique message, the sending event of m is $S(m)$ and the receive event of m is $R(m)$. Note that the receipt of a message occurs at some time later than the arrival of that message at the receiving machine. The term arrival indicates that the message is physically within the buffers of the destination machine. When the message is transfered to the data space of the process and the process is informed of that transfer, the message is received.

In other contexts, send and receive events may have different meanings. For example, if we were given an environment where the notion of shared memory was implemented, the send event may refer to the writing of a message to a well-known location in that shared memory, and the receive event may refer to the reading of that location by the recipient process. However, in all cases, a send event is the locally necessary action that initiates an inter-process communication, and a receive event is the locally necessary action that concludes an inter-process communication.

2.2 Event Order

The framework of time in a single process, sequential system is not present in its distributed counterpart. By definition, processes in a distributed environment execute independently until some communicative interaction is required. Varying processor speeds and internal clock drift account for part of a complexity not found in sequential and tightly coupled parallel systems. In heterogeneous environments, execution of individual instructions are likely to require different numbers of machine cycles, thus adding to the disparity between process executions.

These impediments to serialization of events can, at this point, only be partially overcome. Observations about the relationships of events allow a partial order to be placed on E. The work of Lamport[29] defines these relations.

2.2.1 Causality

Given two events, v and v', we are assured of two facts. First, if both events occur within the same process, then a temporal order must exist between them; i.e., one event must be the first to occur. If event v is executed before event v', then the temporal ordering is given as v *happens before* v'. This relationship is represented as $v \rightarrow v'$. We may also say that the execution of v can possibly have an effect on the execution of v'. For this reason, we say that v *causally precedes* v'. The right arrow can be read as either "happens before" or "causally precedes." The two are equivalent.

Second, transmission of a message also assures us of a relationship between events. If v is the sending event of a message and v' is the receipt of the same message, then $v \rightarrow v'$. In other words, the receipt must happen after the transmission of the message. Together with the intraprocess temporal ordering, the basis of event order in a distributed system is formed.

Definition 2.4 *Event v causally precedes event v', written $v \rightarrow v'$, if*

1. *v occurs temporally prior to v' in the same process,*
2. *v is the sending of a message and v' is the receipt of the same message, or*
3. *there exists event v'' such that $v \rightarrow v''$ and $v'' \rightarrow v'$.*

The causal relation is neither reflexive nor symmetric, but is transitive as exemplified by the third case of the definition. Causality, $\rightarrow \subset E \times E$, is the smallest relation such that a preceding event has a possible effect on its successors. Although the causal relation

transcends the communication paradigm employed, slight alterations in case 2 of the definition must be made if the asynchronous transmission mode is not employed. If synchronous communication, as supported by CSP[26] is used, the sending and receiving events of a message represent a single, distributed rendezvous event. Assuming that v is the sending event and v' is the receiving event, the distributed assignment is given as v/v'. Case 2 of the definition is then written as:

2. *v is the synchronous transmission of a message and v' is the synchronous receipt of the same message then the two events rendezvous and are considered a single, joint event v/v', or*

This notion corresponds to the distributed assignment axiom[26]. Other work by Black, et. al.[5] has formalized the extension of $\rightarrow$ into the synchronous domain by adding additional constraints to those given in the definition. We define a graph **H** to represent the causal relationships of the system's events.

Definition 2.5 *The graph* **H** *represents the transitive closure of the causal relationships derived from the execution of the distributed system. Vertices indicate the execution of a communication event and a directed edge vv' exists if and only if $v \rightarrow v'$.*

From the nature of distributed systems and their execution we can define two properties about **H**. We assume that temporal anomalies are not present in the execution; i.e., there are no cases where both v occurs temporally before v' and v' occurs temporally before v. We also assume that each event in the system is represented by at most one vertex in **H**.

The first property of **H** is derived directly from the transitivity of the causal relation. It states that if a path exists between any two nodes, then an arc must exist between those

two nodes. In other words, the transitive closure of the causal relation is displayed. This is in contrast to the time-space diagram which displays the transitive reduction of the causal relation.

Property 2.1 *All vertices reachable from v are adjacent to v.*

Proof: Assume there exists a path from v to v' such that vv'' and $v''v'$ are edges in **H**.

$$vv'' \Rightarrow v \rightarrow v'' \quad \text{by Def 2.5} \quad (2.1)$$

$$v''v' \Rightarrow v'' \rightarrow v' \quad \text{by Def 2.5} \quad (2.2)$$

$$v \rightarrow v'' \wedge v'' \rightarrow v' \Rightarrow v \rightarrow v' \quad \text{by the transitivity of } \rightarrow \quad (2.3)$$

$$v \rightarrow v' \Rightarrow vv' \quad \text{by (2.3) and Def 2.5} \quad (2.4)$$

■

From our assumption that no temporal anomalies are found in the execution we derive the next property. Each node in **H** represents the occurrence of an event, and arcs represent the possibility of impacting the execution of the tail event. Since a single event cannot precede itself, we do not allow our graph to represent this occurrence. From property 2.1 we know if a path exists from a node, through some other node(s) and back to the original node, then an arc must also exist from the original node to itself. Therefore, no cycles are allowed in **H**.

Property 2.2 *No cycle exists in* **H**.

Proof: Follows directly from the irreflexive, antisymmetric, and transitive properties of the causal relation. ■

The definition of the causal relation provides our first opportunity to divide the events of the distributed system into sets. The events included in a set are *causally equivalent*, that is, all events included in a set possess the same causal relation to events not in the set. Causally equivalent regions are defined by the occurrence of communication events since the inter-process causal relation is based on inter-process communication.

Definition 2.6 *A causally equivalent region, $\mathcal{R}_i^k$, is a contiguous set of non-communication events bounded by the communication event v_i^k and the next succeeding communication event in P_i.*

$$\mathcal{R}_i^k = \{v : \mathcal{P}(v) = i \wedge v \notin M \wedge v_i^k \rightarrow v \wedge \nexists v' \in M : v_i^k \rightarrow v' \rightarrow v\}$$

Note that the definition of a region does not include the communication events in any region. It would be equally precise to include receipt events in the region that immediately follows the receipt. The causal relationship defined with respect to the receive event does not change until the next communication event occurs. In other words, a receive events is causally equivalent to the set of contiguous, local events that immediately follow the receipt. Likewise, a transmission event could be included in the preceding region. However, the definition is designed to simplify the execution of a distributed system to the execution of communication events interleaved by causally equivalent regions which we will model as single entities.

2.2.2 Concurrency

Events that are not causally related can have no effect on each other. Given two events, v and v', if neither causally precedes the other, then their order is undefined. This absence

of order is defined as the *concurrent*[29] relation between events. Two executions of the same distributed system with identical inputs may produce different temporal orderings of concurrent events regardless of their temporal order in a single execution. It is not precise to refer to events that are concurrent as "*happening at the same time.*" Instead, concurrency implies that the events can be executed in any order.

Definition 2.7 *Events v and v' are concurrent, written $v \| v'$, if and only if $v \not\rightarrow v'$ and $v' \not\rightarrow v$.*

As with causality, the concurrent relation is defined on the cross product of system events: $\| \subset E \times E$. Notice that the concurrent relation is not transitive. For example, consider the three process scenario shown in figure 2.1 where P_0 sends a message to P_1 and

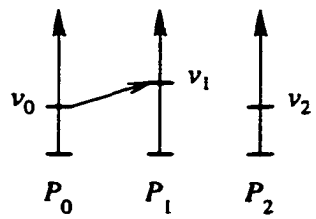


Figure 2.1: Concurrency is not transitive

P_2 executes a local computation. If these are the only events in the system, then we have $v_0 \| v_2$ and $v_2 \| v_1$ since no interaction was made between P_2 and either of the other processes. However,

$$v_0 \| v_2 \wedge v_2 \| v_1 \not\Rightarrow v_0 \| v_1$$

since we know by definition 2.4 that $v_0 \rightarrow v_1$.

Display techniques generally succeed in detailing an accurate representation of the causality present in a distributed system execution. In effect, $\mathbf{H}$ is correctly characterized. However, the complexity of the concurrent relation and its non-transitive nature provide the fodder for inaccurate representations. The graph $\bar{\mathbf{H}}$ is the undirected complement of $\mathbf{H}$ and represents the concurrent relations of the execution.

Definition 2.8 *The graph $\bar{\mathbf{H}}$ represents the concurrent relationships derived from the execution of the distributed system where the vertices indicate the execution of a communication event and an undirected edge joins vertices v and v' if and only if $v \| v'$.*

In both $\mathbf{H}$ and $\bar{\mathbf{H}}$, the relationships between vertices are explicit. The transitive causal relationships are shown as directed arcs in $\mathbf{H}$ instead of indirect paths as in the time-space diagram. This explicit display allows $\bar{\mathbf{H}}$ to be constructed directly from $\mathbf{H}$ by connecting only those vertices in $\bar{\mathbf{H}}$ that were disjoint in $\mathbf{H}$. Since transitivity is explicit in $\mathbf{H}$, we are assured that disjoint vertices are not causally related. The lack of causality indicates the presence of concurrency and, therefore, an undirected arc in $\bar{\mathbf{H}}$.

We can identify the subgraphs of the concurrency graph that are legitimate. As shown in the following property, subgraphs of $\bar{\mathbf{H}}$ with girth[1] greater than four are not possible. It may be possible to find a simple cycle of length greater than four in $\bar{\mathbf{H}}$. However, there will always be another simple cycle of length four or less between the same vertices.

Property 2.3 *There does not exist an induced[2] subgraph $\bar{h} \in \bar{\mathbf{H}}$ such that the girth of that subgraph is greater than four, i.e., $\forall \bar{h} \in \bar{\mathbf{H}} g(\bar{h}) \leq 4$.*

[1] The *girth* of a graph, $g(H)$, is the shortest cycle in H.
[2] *An* induced *subgraph h of $\mathbf{H}$ is one that contains all edges vv' of $\mathbf{H}$ if both v and v' are vertices in h.*

Proof: We prove the property by contradiction. Assume that their exists a subgraph of $\bar{\mathbf{H}}$, $\bar{h}$, such that $g(\bar{h}) = x$ and $x \geq 5$. Let h be the subgraph of $\mathbf{H}$ such that $\bar{h}$ is the undirected complement of h. Also let the vertices of $\bar{h}$ be numbered (using arithmetic modulo the number of vertices in $\bar{h}$) such that vertex $\bar{v}_i$ is adjacent only to vertices $\bar{v}_{i-1}$ and $\bar{v}_{i+1}$. We select five vertices of $\bar{h}$ as shown in figure 2.2.

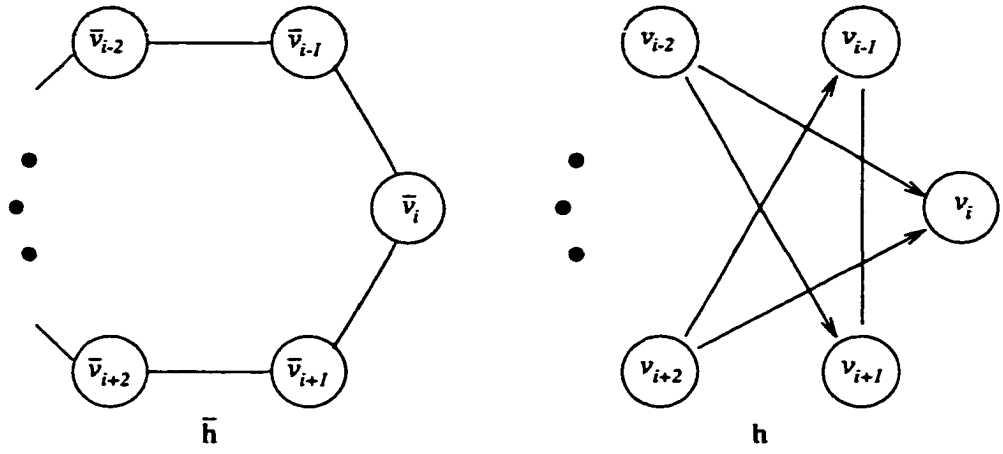


Figure 2.2: Subgraphs of $\bar{\mathbf{H}}$ with girth ≥ 5 are not possible

The absence of an edge in $\bar{h}$ from $\bar{v}_i$ to $\bar{v}_j$ implies the existence of a directed edge in h between v_i and v_j. Although we cannot discern from $\bar{h}$ the direction of the arcs in h, we can make some observations about their relative direction.

If arcs are placed between the selected five vertices such that each vertex has one incoming arc and one outgoing arc, a five vertex cycle is created. Since cycles are not allowed in $\mathbf{H}$ (by property 2.2) and thus not allowed in h, we know this cannot be the case. Therefore, at least one vertex must have two incoming arcs. Let v_i be the vertex in h with two incoming arcs. That is, both $v_{i-2}v_i$ and $v_{i+2}v_i$ are in h.

If the direction of the arc between v_{i-2} and v_{i+1} is $v_{i+1}v_{i-2}$, then we violate the original constraints of $\bar{h}$. Specifically, the existence of arcs $v_{i+1}v_{i-2}$ and $v_{i-2}v_i$ imply, by proposition 2.1, the existence of arc $v_{i+1}v_i$ which is known to be absent in h since $v_i \| v_{i+1}$ in $\bar{h}$. We must assume that the direction of the arc between v_{i-2} and v_{i+1} is $v_{i-2}v_{i+1}$. A similar argument is made for the arc between v_{i-1} and v_{i+2} forcing the direction of the arc to be $v_{i+2}v_{i-1}$.

We now consider the direction of the remaining arc between v_{i-1} and v_{i+1}. If we let the direction of the arc be $v_{i-1}v_{i+1}$, then we transitively create the arc $v_{i+2}v_{i+1}$ which is not present in h. If we instead let the direction be $v_{i+1}v_{i-1}$, then the arc $v_{i-2}v_{i-1}$ is created through transitivity. This arc is also not present in h. Therefore, it must be the case that $v_{i-1} \not\rightarrow v_{i+1}$ and $v_{i+1} \not\rightarrow v_{i-1}$. By definition, $v_{i-1} \| v_{i+1}$. Since concurrent vertices are connected in $\bar{h}$, there exists a cycle between the selected five vertices that does not include v. We conclude that the girth of $\bar{h}$ is $x - 1$ which contradicts the original assumption. ■

Events that are concurrent can, by definition, have no effect on the execution of each other. In some circumstances it is possible for concurrency to be problematic in the distributed system. Assume that both events v and v', where $\mathcal{P}(v) \neq \mathcal{P}(v')$, represent the entry into a critical section and the events are not causally related. It is possible, though not certain, that their occurrence could have been simultaneous. Given a system with concurrently related critical sections, multiple execution tests may not reveal a critical section violation. However, it could be the case that after the software has been put into production, the right circumstances occur to cause the concurrent events to happen simultaneously, violating the mutual exclusion property of critical sections.

The relationship between causality and concurrency is evident from their definitions. The following three properties refer to the execution depicted in figure 2.3 and provide the groundwork for latter use. The first property states that the range of concurrency of an

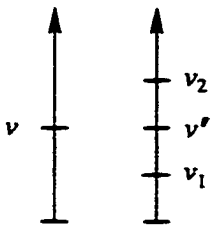


Figure 2.3: The relationship between causality and concurrency

event is contiguous. That is, if we are given an anchor event that is concurrent to both the beginning and ending events of a causal chain, then the anchor is also concurrent to all other events on that chain.

Property 2.4 $\forall v_1, v_2, v : v_1 \rightarrow v_2 \wedge v_1 \| v \wedge v_2 \| v \Rightarrow \forall v' : v_1 \rightarrow v' \rightarrow v_2, v' \| v$

Proof: Assume that v' and v are not concurrent. Then either $v' \rightarrow v$ which implies $v_1 \rightarrow v$ through transitivity, or $v \rightarrow v'$ which implies $v \rightarrow v_2$. In either case, we contradict the assumption. ■

Each process in the system has a set of contiguous events that are concurrent to an anchor event v. Each set except the one from $\mathcal{P}(v)$ may be empty. The union of these sets is referred to as the *concurrent region* with respect to v. Definition 2.9 follows directly from property 2.4.

Definition 2.9 ***A concurrent region*** *with respect to v, CR_v, is the set of events that are*

concurrent to v.

$$CR_v = \{v' \in E : v' \| v\}$$

In the second property, we are again given an anchor event and a causal chain. This time, we are only assured that the anchor event is concurrent to the last event in the chain. This is enough, however, to imply that the anchor does not precede the event beginning the chain.

Property 2.5 $\forall v_1, v_2, v : v_1 \rightarrow v_2 \wedge v_2 \| v \Rightarrow v \not\rightarrow v_1$

Proof: Assume that $v \rightarrow v_1$. Then by transitivity, we have $v \rightarrow v_2$ which contradicts the assumption. ■

The third proposition is similar to the second. In this case we assume concurrency between the anchor event and the event beginning the chain. We can deduce that the anchor event cannot causally follow the chain ending event.

Property 2.6 $\forall v_1, v_2, v : v_1 \rightarrow v_2 \wedge v \| v_1 \Rightarrow v_2 \not\rightarrow v$

Proof: Assume that $v_2 \rightarrow v$. Then by transitivity, we have $v_1 \rightarrow v$ which contradicts the assumption. ■

2.2.3 Partial Order

The causal relation defines a partial order among all events in the distributed system[29]. For any two events v and v', given that $v \rightarrow v'$, v is ordered before v' in the partial order. If these two events are not causally related, $v \| v'$, then there exists no order relative to the

two events. Hence, the ordering of E is only partial if there exists any concurrency in the system.

Note that the accepted definition of a partial order relation requires that the relation possess three basic properties: reflexivity, antisymmetry, and transitivity[19, 20, 35]. As stated previously, the causal relation is non-reflexive since an event cannot precede itself. For this reason, the causal relation has been dubbed an *irreflexive partial order*[29] or the *reflexive reduction*[24] of the partial order. In the remainder of this work, the relation will simply be referred to as a partial order.

2.3 Clocks and Time

Given a global clock where each process has identical notions of the current time, a total ordering of events is possible. In this case, events can be ordered temporally. Concurrency will be ignored given the sequentialized view of the execution. However, the concurrent nature of the problem still exists. Our concerns regarding a seemingly correctly operating program failing as a direct result of timing errors are still valid. The temporal ordering of one execution may provide a false sense of ordering based on random temporal occurrences. We instead rely on the partial ordering based on causality. To model this order, we employ a technique developed independently by Mattern[32] and Fidge[14]. Their technique models the causal order of execution without introducing false relationships based on temporal occurrences.

2.3.1 Vector Time

Each process maintains a local vector which represents the current "causal clock" value. The vector is of length N. The vector at P_i is $\tau_i = (\tau_i[0], \tau_i[1], \ldots, \tau_i[N-1])$ with initial value $\forall k \neq i : \tau_i[k] = 0$, $\tau_i[i] = 1$. Each significant[3] event v in P_i causes $\tau_i[i]$ to be incremented. The updated value of τ_i is then associated with the event, and the vector time of that event is $\tau(v)$. Note that $\tau_i[i]$ is an accurate indication of the number of significant events that have occurred on P_i.

Message transmission events, both sends and receives, require further updates to maintain the isomorphism between vector time and causality. The vector time of the sending event is appended to the outgoing message m as $\tau(m)$. On receipt of message m at P_i, the local vector time of P_i is updated by a two-step process. First, the entire local vector is set to the component-wise maximum of the local vector and the vector attached to the message m. Then the i^{th} element of the local vector time is incremented to reflect the occurrence of the receive event. It has been proven that vector times updated in this manner are isomorphic to the causal relationships of the underlying distributed system[39].

Definition 2.10 *A vector clock of P_i, τ_i, is isomorphic to the causal relationships in the represented system if τ_i is initialized as $\forall_{j \neq i} \tau_i[j] = 0$, $\tau_i[i] = 1$ and updates are as follows.*

1. *If the event is a local computation:*

   (a) $\tau_i[i]++$

   (b) the event is stamped with τ_i

[3]The definition of *significant* is intentionally left nebulous. The events concerning the software engineer will include the communication events but may also include other events.

2. *If the event is a send event:*

   (a) $\tau_i[i]++$

   (b) *the event and the message are stamped with* τ_i

3. *If the event is a receipt event of message* m*:*

   (a) $\forall j, \tau_i[j] = \max(\tau_i[j], \tau(m)[j])$

   (b) $\tau_i[i]++$

   (c) *the event is stamped with* τ_i

The vector times of any two events can be compared to identify any causal relationships between the events. As shown in the following definition, event v occurs before event v' if and only if $\tau(v)[i] \leq \tau(v')[i]$ for all i and the times are not identical. On the other hand, if comparisons show $\tau(v)[i] < \tau(v')[i]$ for some i and $\tau(v)[i] > \tau(v')[i]$ for other i, then the times are incomparable and the events are concurrent.

Definition 2.11 *Given two events,* v *and* v'*, and their vector times,* $v \rightarrow v'$ *if and only if* $\tau(v) < \tau(v')$ *and* $v \| v'$ *if and only if* $\tau(v) \| \tau(v')$*, as expressed in the following expressions.*

- $\tau(v) \leq \tau(v')$ *if and only if* $\forall i : \tau(v)[i] \leq \tau(v')[i]$
- $\tau(v) < \tau(v')$ *if and only if* $\tau(v) \leq \tau(v') \wedge \tau(v) \neq \tau(v')$
- $\tau(v) \| \tau(v')$ *if and only if* $\tau(v) \not\leq \tau(v') \wedge \tau(v') \not\leq \tau(v)$

Vector clocks may initially seem unnecessarily bulky. However, work by Charron-Bost[8] has proven that a vector of length N is both necessary and sufficient to accurately represent

the causality of an N-process distributed system. This work was based on the assumption that no prior information concerning the communication patterns of the system is known. We believe that this is a reasonable assumption that only generalizes the proof.

Although definition 2.10 is the traditional definition of a vector clock, we can alter the definition to allow a better graphic interpretation by replacing rule 3.(b) as follows.

3. *(b)* $\tau_i[i] = \max(\tau_i) + 1$

The altered vector clock retains the property that the local component of the vector clock is larger than the local component of the vector clock preceding it in the same process. We have added the property that the local component is also larger than all components of causally preceding vector clocks. Theorem 2.1 proves that the altered vector clock remains isomorphic to causality.

Theorem 2.1 *The modified vector clock is isomorphic to causality.*

Proof: The modified vector clock differs from the original clock only in the local component of a receive event. Let $v_i \in E_i$ be the receipt event of message m, and let $v_j \in E_j$ be the send event of message m. Also let $v_i' \in E_i$ be the event in P_i that immediately precedes v_i. That is, $\nexists v : v_i' \rightarrow v \rightarrow v_i$.

From vector clock update rule 2(b), we know that $\tau(m) = \tau(v_j)$. Using the original vector clock update rules, rule 3(a) sets the local component of $\tau(v_i)$ to the maximum of $\tau(v_i')[i]$ and $\tau(m)[i]$ and then rule 3(b) increments the local component by one. Therefore, we have $\tau(v_i)[i] > \tau(v_i')[i]$ and $\tau(v_i)[i] > \tau(v_j)[i]$.

The modified vector clock update rules only differ in rule 3(b) which sets the local component of v_i as follows.

$$\tau(v_i)[i] = \max(\forall j : \tau(m)[j], \tau(v_i')[i]) + 1$$

This assignment insures that $\tau(v_i)[i] > \tau(v_i')[i]$ and $\tau(v_i)[i] > \tau(v_j)[i]$.

While the increase in the local component of the vector time is no longer always one, the local component does increase at every event. We have shown that the sole difference between the original and modified vector clocks does not alter the relationships between the altered components. Since the original vector clock is isomorphic to causality, the modified vector clock must also be isomorphic to causality. ■

We can let the N-component vector clock represent coordinates on an N-dimensional graph. Each axis of the graph represents execution in a process with time increasing with distance from the origin. For simplicity, we present only an $N = 2$ case. Figure 2.4 shows a simple two process execution and the graph of the vector times of the executed events.

P_0		P_1	
send m_0	[1,0]	send m_2	[0,1]
send m_1	[2,0]	recv m_0	[1,2]
recv m_2	[3,1]	recv m_1	[2,3]
send m_3	[4,1]	send m_4	[2,4]
recv m_4	[5,4]	recv m_3	[4,5]

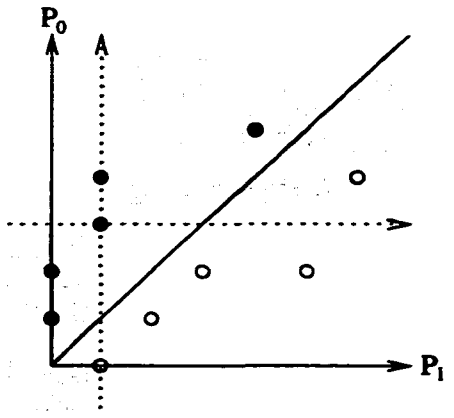


Figure 2.4: Graph interpretation of vector clock

We now make two observations about our altered vector time and the concurrency graph. In theorem 2.2 we show that the closest axis to the coordinates of an event corresponds to the process executing the event. For example, in the two dimensional case, the events executed by process P_0 are drawn to the P_0 side of the diagonal.

Theorem 2.2 *For all events v_i with vector clock τ_i, the distance from the display coordinates of v_i to the P_i axis is less than the distance from the display coordinates of v_i to the $P_{j\neq i}$ axis.*

Proof: We compute the distance from τ_i to the closest point on the P_i axis, $(0, ..., \tau_i[i], ..., 0)$. Each component of the sum of squares is $\tau_i[j] - 0$ except the i^{th} component, which is $\tau_i[i] - \tau_i[i]$. In the following equation, $\tau_i[i]^2$ is removed from the summation.

$$\Delta_i = \sqrt{\tau_i[0]^2 + \tau_i[1]^2 + \cdots + \tau_i[i-1]^2 + \tau_i[i+1]^2 + \cdots + \tau_i[n-1]^2}$$

A similar formula is constructed for the distance from τ_i to the $P_{j\neq i}$ axis. The j^{th} component is zero in this case. Therefore, $\tau_i[j]^2$ is removed from the following equation.

$$\Delta_j = \sqrt{\tau_i[0]^2 + \tau_i[1]^2 + \cdots + \tau_i[j-1]^2 + \tau_i[j+1]^2 + \cdots + \tau_i[n-1]^2}$$

Let d_{ij} be the sum of the squares of all components except i and j.

$$d_{ij} = \tau_i[0]^2 + \cdots + \tau_i[n-1]^2 - \tau_i[i]^2 - \tau_i[j]^2$$

Substituting d_{ij} into the previous two equations, we have

$$\Delta_i = \sqrt{d_{ij} + \tau_i[j]^2}$$

$$\Delta_j = \sqrt{d_{ij} + \tau_i[i]^2}$$

From vector clock update rule 3.(*b*) we know that $\tau_i[i] > \tau_i[j]$ for all $j \neq i$ making $\Delta_i < \Delta_j$. Therefore, the distance from τ_i to the P_i axis is less than than the distance from τ_i to any other axis. ■

Suppose the origin of the graph is translated to the display coordinates of a particular event. In figure 2.4 we have arbitrarily chosen the receipt of message m_2 in process P_0 and have drawn the translated axes as dashed lines. With respect to the translated origin of a two processes system, all events displayed in or on the border of quadrant I *happen after* the selected event. Likewise, all events displayed in or on the border of quadrant III *happen before* the selected event. Events that are displayed in neither quadrant I or III are concurrent to the selected event. The figure shows that events in the unshaded areas are the concurrent events while those in the shaded areas are causally related to the receipt of m_2. Both the first two events of P_0 and the first event of P_1 *happen before* the receipt of m_2.

We can describe this visual attribute using vector clock differences. If the difference of the vector clocks, and therefore the display coordinates, of events v and v' is completely non-positive, then the relationship must be v *happens before* v'. Similarly, if the difference is all non-negative the relationship must be v *happens after* v'. This result is a direct application of definition 2.11 in the visual domain.

Theorem 2.3 *Given two events v and v', v is displayed no further from any axis than v' if and only if $v \rightarrow v'$.*

Proof: Assume that $v \rightarrow v'$. We know from theorem 2.1 that $\tau(v) < \tau(v')$. That is,

$$\forall j, \tau(v)[j] \leq \tau(v')[j] \wedge \exists i : \tau(v)[i] < \tau(v')[i].$$

Since the event is displayed at the coordinates specified by it's vector clock, v must be displayed no further than v' from any axis.

Assume that v is displayed no further from any axis than v'. Then all components of the display coordinates of v must be less than or equal to the display coordinates of v'. Therefore,

$$\forall j, \tau(v)[j] \leq \tau(v')[j].$$

Furthermore, if v and v' are separate events they must differ in some component.

$$\forall j, \tau(v)[j] \leq \tau(v')[j] \wedge \exists i : \tau(v)[i] \neq \tau(v')[i].$$

From definition 2.11, we know that $v \rightarrow v'$. ■

Obviously this type of display is limited. As the number of constituent processes increases, so does the dimensionality of the display. It is an interesting and solvable mental exercise to extend the technique into three dimensions. In chapter 4 we will expand this idea to construct an accurate display.

2.4 Abstraction

Even in a simple distributed system with minimal concurrency we are faced with an over abundance of information. To reduce the cognitive strain on the software engineer, this

information is boiled down to a more easily absorbed form. We have seen in a previous section that multiple consecutive local computation events can be modeled, without loss of accuracy, as a single local computation event. This type of abstraction is one of several that have been suggested in the literature.

Although abstraction plays an important role in the understanding of the system, its application to the display of concurrency is limited. If the software engineer needs to know what events in each process are concurrent to a selected event, abstraction techniques that hide events in larger, more complicated structures may prevent that knowledge from being accessible.

2.4.1 Process Abstraction

The modular design methodology is the driving principle behind *process abstraction*[10]. Each process is created such that it is composed of cooperating independent modules. Each module has *interface points* through which it communicates with other modules. The obvious advantage to writing programs in modules is that it allows the software engineer to concentrate on a single module without regard to the entire system. Once the module is operating correctly, it need only be connected via the interface points to the remainder of the system. A visualization tool for such an abstraction may create a display similar to the one in figure 2.5.

When the communicating modules of a process have been constructed and tested, they can be abstracted into a meta-module with its own interface points. Repetition of the technique is claimed to yield an arbitrarily simplified view of the system. However, the interaction among modules is not displayed. While each module may operate correctly in

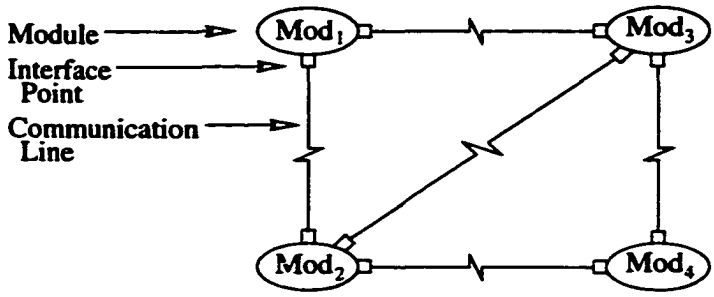


Figure 2.5: Process abstraction

isolation, the system may fail is inter-module relationships are ignored.

2.4.2 Behavioral Abstraction

Two phases, *filtering* and *clustering*, compose *behavioral abstraction*[3]. We previously stated that we would consider only significant events in the display. This is also the goal of filtering. All events deemed insignificant are ignored just as we ignored the multiple consecutive local computation events. Insignificance is not limited to the local events. It can be composed of events from the execution of a portion of the process that is thought to be correct. From the other point of view, we can select certain sections of the source code from which events are considered significant. With care, entire processes may be filtered out as insignificant.

After filtering has removed the initially determined insignificant events the clustering phase is begun. Here events are clustered together to form meta-events. When both phases are complete, the process can be repeated. This time the filtering will attempt to remove unwanted meta-events and the clustering will create meta-meta-events. The process is repeated until a significantly simplified view of the system is achieved.

2.4.3 Molecular Abstraction

A somewhat different approach to abstraction was given by Ahuja, Kshemkalyani, and Carlson[1]. This method groups corresponding send and receive events into *atoms* of execution. Multiple atoms are combined to form *molecules* such that if the send and receive events of the atom have different causal relationships to the molecule, then the atom is included in the molecule. Figure 2.6 shows the molecular structure of an execution of a six process system. Horizontal arrows represent the passing of time in a single process and diagonal arrows represent message transferral. The molecules are shown as events inclosed in a dashed figure.

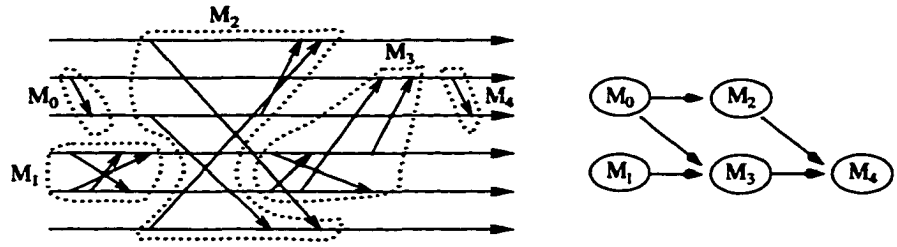


Figure 2.6: Molecular abstraction

Note that molecules M_0 and M_4 are also atoms. Since they are set apart from the other communications, they are not included with other atoms in a molecular structure. The figure also shows the relationships of the molecules after the first abstraction. In this figure, the ovals represent molecules and the arrows show the direction of causal flow.

2.4.4 Event Abstraction

The condensation of several primitive events into a single representative event was first introduced by Cheung[10] in his dissertation. As shown in the definition of causally equiv-

alent regions, this may be easily accomplished if all the primitive events to be combined are consecutive, local events. However, care must be taken to maintain the irreflexive and transitive properties of causality if communication events are included in the abstraction. According to Cheung, any group of events can be combined as long as these properties are maintained and *event abstraction* may be applied multiple times until a significantly reduced graph is constructed.

In her concurrency map, Stone[45] uses a modified form of event abstraction. Although not explicitly stated in her paper, a single abstraction creates *blocks* of events that are similar to our causally equivalent regions. In the next chapter we will present an algorithm for creating event blocks and explore the concurrency map in more detail.

Chapter 3

The Concurrency Map

Stone's concurrency map[45] is a postmortem display technique for distributed systems. The concurrency map uses a single static diagram to represent both the causal and concurrent relationships between all events in the system. Event abstraction is used to create event blocks that are placed in a two dimensional grid. The relative position of two blocks indicates the relationship between them. Both causality and concurrency are shown explicitly, which is in contrast to the implicit display of concurrency in the time-space diagram.

3.1 Map Creation

Although not explicitly stated in her paper, Stone uses event abstraction to group causally equivalent events into groups called *blocks*. A block differs from a causally equivalent region in that communication events are included in a block. A send event is included in the block of events preceding it while a receive event begins a new block.

Algorithm 3.1 formalizes the construction of blocks as used in the concurrency map. A single pass is made through the list of events to create the blocks. This algorithm cannot be repeated to yield a more compact representation.

```
for i = 0...N-1 do                          /* iterate over each process */
    k = 0                                   /* start with block 0 */
    B_i^k = {}                              /* block 0 is initially empty */
    while more v ∈ E_i are found do
        if v ∈ S                            /* this event is a send event */
            B_i^k = B_i^k ∪ {v}              /* add event to current block */
            k = k + 1                       /* begin new block */
            B_i^k = {}                      /* new block is initially empty */
        elseif v ∈ R                        /* this event is a receive event */
            k = k + 1                       /* begin new block */
            B_i^k = {v}                     /* this event is first event in block */
        else                                /* this event is a local event */
            B_i^k = B_i^k ∪ {v}              /* add this event to the current block */
        fi
    od  /* while */
od  /* for */
```

Algorithm 3.1: Creating **blocks** of events through event abstraction

Placement of blocks inside the grid is based on two factors: the process that executed the block and the length of the causal chain leading to that block. Blocks that are executed by process P_i are placed in column i in the order they were executed. The first block is placed at the top of the grid. Blocks are not allowed to overlap. Further restrictions on vertical placement of blocks is based on causal relationships.

Rows display the relationships between blocks occurring in different processes and are numbered from 0 at the top of the concurrency map. Interprocess relationships are identified by the vertical separation of blocks executed by different processes into different rows. If two blocks in different columns are at least partially contained in a common row, then the the blocks are concurrent. On the other hand, if two blocks, B and B', are positioned without a common row, then a causal relationship exists between them. Furthermore, if B is located above B' (B is in a lower numbered row than B'), then $B \rightarrow B'$.

Definition 3.1 *The* **image** *of a block in the concurrency map, $\mathcal{I}(B)$, is the set of rows on which that block is mapped.*

Definition 3.2 *The minimum (topmost) row of $\mathcal{I}(B)$ is $\lfloor B \rfloor$.*

Definition 3.3 *The maximum (bottommost) row of $\mathcal{I}(B)$ is $\lceil B \rceil$.*

The original definition of the concurrency map was given in vague prose. For example, the construction algorithm was given as the single statement,

> ... any successor event that is reached by a chain of interprocess dependencies of at most n is in row $n + 1$.[45]

We provide a more formal definition of the concurrency map based on the definition of the image of a block and the original paper.

Definition 3.4 *A* **concurrency map** *is a two dimensional grid representation of the execution of a distributed system where*

1. *event abstraction (algorithm 3.1) is used to construct causally equivalent blocks,*
2. *blocks executed by process P_i are placed in row i in non-overlapping temporal order such that $B \rightarrow B'$ if and only if $\lceil B \rceil \leq \lfloor B' \rfloor$,*
3. *causally related blocks from different processes are placed such that $B \rightarrow B'$ if and only if $\lceil B \rceil < \lfloor B' \rfloor$, and*
4. *concurrent blocks are placed such that if $B \| B'$ then $\mathcal{I}(B) \cap \mathcal{I}(B') \neq \emptyset$.*

The following corollary follows directly from definition 3.4. In fact, it is simply a restatement of the final item in terms of the minimum and maximum of the images of blocks. These terms will be utilized in the proofs to follow.

Corollary 3.1 *$B_i^k \| B_j^l$ if and only if $\lceil B_i^k \rceil \geq \lfloor B_j^l \rfloor \wedge \lfloor B_i^k \rfloor \leq \lceil B_j^l \rceil$.*

3.1.1 An Example

As a demonstration of the utility of the concurrency map, consider the case of a multiprocess system with a single process serving as a work server. Process P_0 sends a unit of work to each other process. These processes independently compute results which are returned to P_0 to be assembled. When all processes have reported, new work loads are calculated and sent, restarting the procedure. The algorithms for the server and clients are shown in algorithm 3.2.

```
repeat                                    repeat
   foreach 1 ≤ j < N                         receive(work) from P0
      send(work) to Pj                       compute()
   repeat                                    send(response) to P0
      receive(response) from any P        forever
   until all P have responded
forever
```

Algorithm 3.2: Algorithms for server and computation processes

The block relationships become obvious when viewed as a concurrency map. Figure 3.1 shows the map constructed from a trace of a four process system. Since the intersection of the images of P_2's block and the [recv m_4] block of P_0 is not empty, we can correctly deduce a concurrent relationship between them. However, the disjoint images of the block in P_3 and the [recv m_4] block of P_0 imply a causal relationship that is present in the trace due to message m_3 and the temporal occurrence of the receipts of P_0.

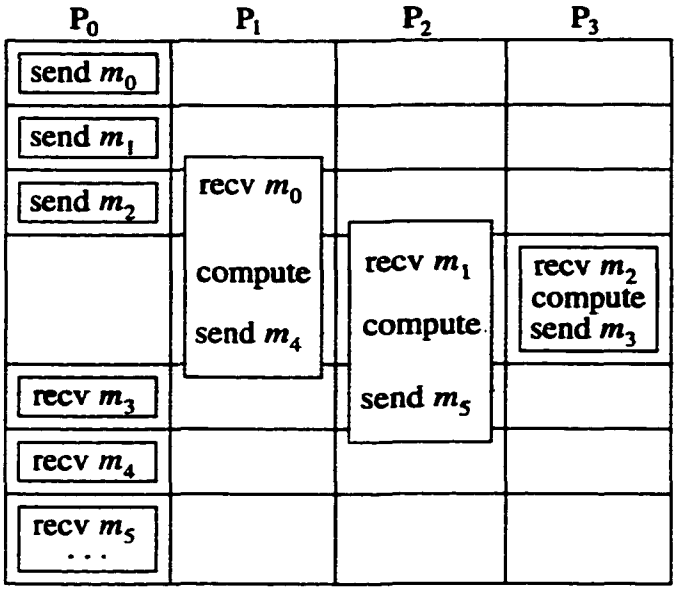


Figure 3.1: Concurrency map of server/client execution

In this concurrency map, both causality and concurrency are explicitly shown. But in a more complicated example, one without global constraints, problems arise. In these cases, the greatly simplified view of the system's execution is possible only at the expense of sacrificed accuracy.

3.1.2 The Inaccuracy

Assume we are given a three process system. Suppose each process is a sequence of events that follows the pattern {*compute, communicate, compute,* ... } where communication is either a message receipt or transmission. In Stone's paper, we are given an example execution which we will use to demonstrate the problem. The traced events are shown in figure 3.2. Construction and placement of blocks into the concurrency map grid results in map also shown in figure 3.2.

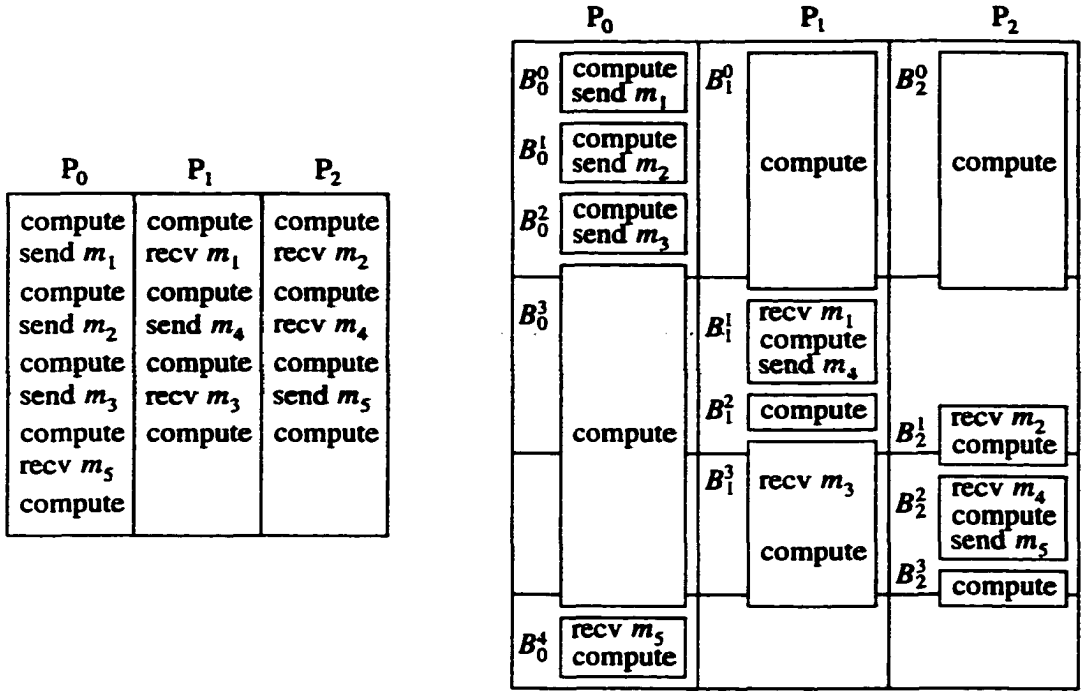


Figure 3.2: Event execution and the derived concurrency map

On the surface, the concurrency map appears to identify the relationships between the blocks in a manner conducive to understanding the system. Closer examination reveals a subtle problem. Some causal constraints present in the concurrency map are not present in the trace from which it was derived. For example, since blocks B_0^1 and B_1^1 are shown separated into different rows, a causal link is implied; i.e., $B_0^1 \rightarrow B_1^1$. In the trace, however, it is the case that $B_0^1 \| B_1^1$. A naive attempt to correct the situation may move B_0^1 down to show the concurrency with B_1^1. This would require that B_2^1 also be moved down to maintain the display of the causality between B_0^1 and B_2^1, thus preventing the display of $B_1^0 \| B_2^1$. In fact, no matter how cleverly we place blocks in the map, it is not possible to accurately represent the causal relationships of the traced system without including spurious causality.

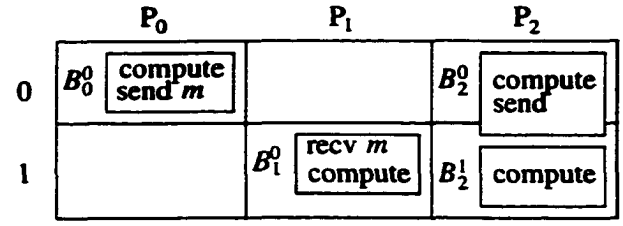


Figure 3.3: Simplified concurrency map of $N = 3$ process system

The difficulty stems from the block relationships exemplified by the simplified concurrency map of figure 3.3 involving three processes. Note that there must exist more blocks than those shown for this situation to be possible. Specifically, there must exist a receive event in either P_0 or P_1 for the message sent from P_2. We consider only those blocks needed for identification of the problem. Here we have the transmission of a message from from P_0 to P_1. In addition, two blocks in P_2 are each concurrent to both the send and receive blocks. Since the causally related blocks must be separated by a row division and blocks in the same process cannot overlap, all the concurrency cannot be displayed. Theorem 3.1 proves this impossibility.

Theorem 3.1 *The constraints present in figure 3.3 cannot be accurately represented as a concurrency map.*

Proof: We proceed by contradiction. Assume the concurrency map can be accurately constructed.

$$\lfloor B_1^0 \rfloor \leq \lceil B_2^0 \rceil \qquad \text{corollary 3.1 and } B_1^0 \| B_2^0 \tag{3.1}$$

$$\lfloor B_2^1 \rfloor \leq \lceil B_0^0 \rceil \qquad \text{corollary 3.1 and } B_2^1 \| B_0^0 \tag{3.2}$$

$$\lceil B_0^0 \rceil < \lfloor B_1^0 \rfloor \qquad \text{definition 3.4 (3) and } B_0^0 \rightarrow B_1^0 \tag{3.3}$$

$\lfloor B_2^1 \rfloor < \lceil B_2^0 \rceil$	(3.1), (3.2) and (3.3)	(3.4)
$\lceil B_2^0 \rceil \leq \lfloor B_2^1 \rfloor$	definition 3.4 (2) and $B_2^0 \rightarrow B_2^1$	(3.5)
False	(3.4) and (3.5)	(3.6)

■

Distributed systems in which this type of causal/concurrent pattern is found are common. Suppose we are given system of at least three processes. If two processes concurrently send a message to a third process, the pattern will be present in the trace. One would be hard-pressed to construct a realistic system of at least three communicating processes which does not contain this pattern. Referring back to the concurrency map of figure 3.2, blocks B_0^1, B_1^0, B_1^1 and B_2^1 have this relationship.

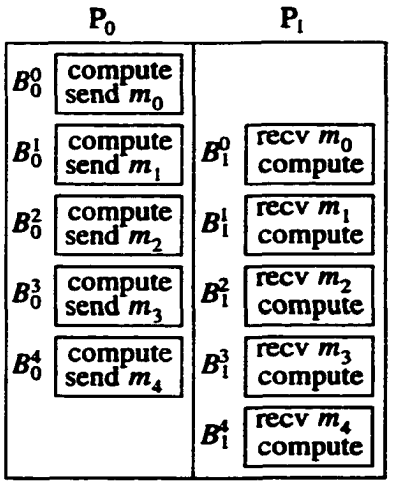


Figure 3.4: Producer/Consumer with 2 processes

Another fault in the concurrency map is revealed if we are given a two process system functioning in a producer/consumer fashion as shown in figure 3.4. Assume we are

attempting to represent a two process system where P_0 sends several consecutive messages to P_1 which, in turn receives those messages. As before, a map for this system cannot be constructed.

Before we prove that the concurrency map for this scenario cannot be accurately constructed, we first prove lemma 3.1. The lemma makes a strong statement about the placement of two blocks, B_i^k and B_j^l, if 1) the B_i^k and B_j^l are concurrent, and 2) the block preceding B_i^k is concurrent to the block following B_j^l. If a block does not precede B_i^k, then we consider the relationship of the process initial event to the block following B_j^l.

Lemma 3.1 *If* $B_i^k \| B_j^l \wedge B_i^{k-1} \| B_j^{l+1}$ *then* $\lfloor B_i^k \rfloor = \lceil B_j^l \rceil$.

Proof: We proceed by contradiction. Assume that $\lfloor B_i^k \rfloor \neq \lceil B_j^l \rceil$.

$\lfloor B_i^k \rfloor \leq \lceil B_j^l \rceil$	corollary 3.1 and $B_i^k \| B_j^l$	(3.7)
$\lfloor B_i^k \rfloor < \lceil B_j^l \rceil$	(3.7) and assumption	(3.8)
$\lceil B_i^{k-1} \rceil \leq \lfloor B_i^k \rfloor$	definition 3.4 (2) and $B_i^{k-1} \rightarrow B_i^k$	(3.9)
$\lceil B_i^{k-1} \rceil < \lceil B_j^l \rceil$	(3.8) and (3.9)	(3.10)
$\lceil B_j^l \rceil \leq \lfloor B_j^{l+1} \rfloor$	definition 3.4 (2) and $B_j^l \rightarrow B_j^{l+1}$	(3.11)
$\lceil B_i^{k-1} \rceil < \lfloor B_j^{l+1} \rfloor$	(3.10) and (3.11)	(3.12)
$\lceil B_i^{k-1} \rceil \geq \lfloor B_j^{l+1} \rfloor$	corollary 3.1 and $B_i^{k-1} \| B_j^{l+1}$	(3.13)
False	(3.12) and (3.13)	(3.14)

■

Theorem 3.2 proves the impossibility of constructing a concurrency map representation of the two process producer/consumer system shown in figure 3.4. As with the previous example, this communication pattern is not uncommon. Systems need not maintain a strict producer/consumer relationship. The necessary conditions for the theorem to hold require only five or more messages to be sent and received before a return message is sent.

Theorem 3.2 *The constraints present in figure 3.4 cannot be accurately represented as a concurrency map.*

Proof: We proceed by contradiction. Assume that the concurrency map is accurately constructed. Let $S(m_i) \in B_0^i$ and $R(m_i) \in B_1^i$ $(B_0^i \rightarrow B_1^i)$ for all $0 \leq i \leq 4$ as shown in figure 3.4. These are exactly the blocks constructed by algorithm 3.1.

$$\lfloor B_0^3 \rfloor = \lceil B_1^0 \rceil \qquad \text{lemma 3.1 and } B_0^3 \| B_1^0 \wedge B_0^2 \| B_1^1 \qquad (3.15)$$

$$\lfloor B_0^4 \rfloor = \lceil B_1^0 \rceil \qquad \text{lemma 3.1 and } B_0^4 \| B_1^1 \wedge B_0^3 \| B_1^2 \qquad (3.16)$$

$$\lfloor B_0^3 \rfloor = \lfloor B_0^4 \rfloor \qquad \text{(3.15) and (3.16)} \qquad (3.17)$$

$$\lfloor B_0^4 \rfloor = \lceil B_1^1 \rceil \qquad \text{lemma 3.1 and } B_0^4 \| B_1^1 \wedge B_0^3 \| B_1^2 \qquad (3.18)$$

$$\lfloor B_0^3 \rfloor \leq \lceil B_0^3 \rceil \qquad \text{definitions 3.2 and 3.3} \qquad (3.19)$$

$$\lceil B_0^3 \rceil \geq \lfloor B_0^4 \rfloor \qquad \text{(3.17) and (3.19)} \qquad (3.20)$$

$$\lceil B_0^3 \rceil \leq \lfloor B_0^4 \rfloor \qquad \text{definition 3.4 (2) and } B_0^3 \rightarrow B_0^4 \qquad (3.21)$$

$$\lceil B_0^3 \rceil = \lfloor B_0^4 \rfloor \qquad \text{(3.20) and (3.21)} \qquad (3.22)$$

$$\lceil B_0^3 \rceil = \lceil B_1^1 \rceil \qquad \text{(3.18) and (3.22)} \qquad (3.23)$$

$$\lceil B_1^0 \rceil = \lceil B_1^1 \rceil \qquad \text{(3.16) and (3.18)} \qquad (3.24)$$

$\lceil B_1^0 \rceil \leq \lfloor B_1^1 \rfloor$	definition 3.4 (2)and $B_1^0 \rightarrow B_1^1$	(3.25)
$\lfloor B_1^1 \rfloor = \lceil B_1^1 \rceil$	(3.24) and (3.25)	(3.26)
$\lceil B_0^3 \rceil = \lfloor B_1^1 \rfloor$	(3.23) and (3.26)	(3.27)
$\lceil B_0^2 \rceil \geq \lfloor B_1^1 \rfloor$	corollary 3.1 and $B_0^2 \| B_1^1$	(3.28)
$\lceil B_0^2 \rceil \geq \lceil B_0^3 \rceil$	(3.27) and (3.28)	(3.29)
$\lceil B_0^2 \rceil \leq \lfloor B_0^3 \rfloor$	definition 3.4 (2) and $B_0^2 \rightarrow B_0^3$	(3.30)
$\lfloor B_0^3 \rfloor \leq \lceil B_0^3 \rceil$	definitions 3.2 and 3.3	(3.31)
$\lceil B_0^2 \rceil \leq \lceil B_0^3 \rceil$	(3.30) and (3.31)	(3.32)
$\lceil B_0^2 \rceil = \lceil B_0^3 \rceil$	(3.29) and (3.32)	(3.33)
$\lceil B_0^2 \rceil < \lfloor B_1^2 \rfloor$	definition 3.4 (3) and $B_0^2 \rightarrow B_1^2$	(3.34)
$\lceil B_0^3 \rceil < \lfloor B_1^2 \rfloor$	(3.33) and (3.34)	(3.35)
$\lceil B_0^3 \rceil \geq \lfloor B_1^2 \rfloor$	corollary 3.1 and $B_0^3 \| B_1^2$	(3.36)
False	(3.35) and (3.36)	(3.37)

■

We have shown that some common execution sequences cannot be accurately represented as a concurrency map. In the next section we examine the nature of the concurrency map and it's underlying graph interpretations and implications. We will see that a simple subgraph is the cause of its inaccuracies.

3.2 *Quad Ring*

Given the concurrency graph $\bar{\mathbf{H}}$ of an execution, we define a *quad ring* to be the maximally girthed induced subgraph of $\bar{\mathbf{H}}$. From proposition 2.3 we know that the maximum girth of any subgraph of $\bar{\mathbf{H}}$ is four. If we are given four nodes comprising a *quad ring*, B^0, B^1, B^2 and B^3, then $B^0 \| B^1 \| B^2 \| B^3 \| B^0$ and $B^0 \rightarrow B^2$ and $B^1 \rightarrow B^3$. Definition 3.5 formalizes this notion.

Definition 3.5 *A* quad ring *is a maximally girthed, four-node induced subgraph of* $\bar{\mathbf{H}}$, *i.e.*, $\bar{h} \subseteq \bar{\mathbf{H}} : \mathrm{g}(\bar{h}) = 4$.

A *quad ring* can involve events from either two, three or four processes. Since $\bar{\mathbf{H}}$ is a result of concurrency, and a *quad ring* is a subgraph of $\bar{\mathbf{H}}$, we know that a single process cannot produce a *quad ring*. We also know from proposition 2.3 that there are no *quad rings* involving more than four events, so we cannot involve more than four processes in its construction.

Consider a distributed system where process P_i sends a sequence of messages to P_j without an intermediate returned message. This scenario will produce a *quad ring*. Figure 3.5 shows a time-space diagram of an execution that fits the prescribed parameters. Also shown in the figure is the complete concurrency graph for the execution of P_i and P_j with the arcs forming a *quad ring* shown in bold.

It is possible to display some *quad rings* using a concurrency map. For example, the nodes representing blocks B_i^2, B_i^3, B_j^0 and B_j^1 of figure 3.5 can be accurately displayed in a single row. However, if an additional constraint is added, then accuracy is no longer possible.

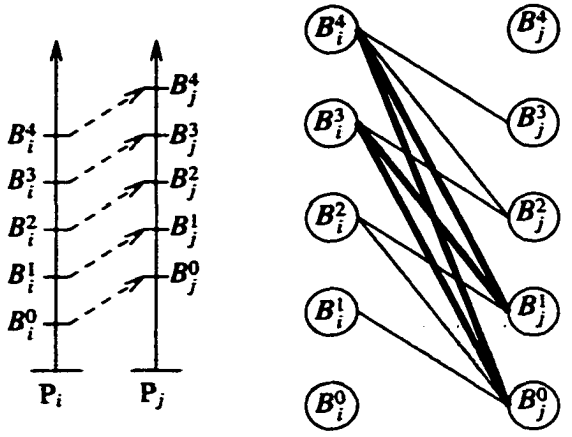


Figure 3.5: A 2 process *quad ring*

The concurrency in a *quad ring* requires that all involved blocks be at least partially within the same row. Suppose that a row division must appear between the blocks of the *quad ring*. Such a circumstance would occur if two of the causally related blocks were from different processes. In this case, we can no longer accurately present the concurrency and the concurrency map fails. When a causal link is present that must be placed within the *quad ring*, we define it to be q^*.

In the two process case, we define a graph q^* to be a *quad ring* where a causal link is found between a block that occurs before the *quad ring* in one process and another block that occurs after the *quad ring* in the other process. The requirement does not limit the size of the distributed system to two processes however. For example, consider the execution shown if figure 3.6 where processes P_j and P_k produce a *quad ring* between B_j^1, B_j^2, B_k^1 and B_k^2. Prior to the *quad ring*, P_j sends a message to P_i and after receiving that message, P_i sends a message to P_k. The message is received at P_k after the *quad ring* has been formed. These two messages form the causal link necessary for a two process q^*.

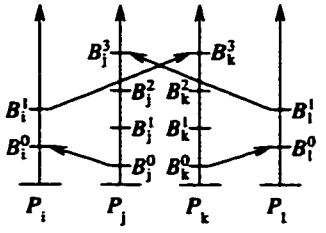


Figure 3.6: A multi-process system producing q^*s

Definition 3.6 formalizes the notion of a two process q^* and includes the relationships between the additional blocks and the *quad ring*. Figure 3.7 shows the simplified concurrency graph for a two-process q^*. Note that two additional *quad rings* are formed in this figure. Namely, B, B^0, B^2 and B^3 form a *quad ring* and B^0, B^1, B^3 and B' form a *quad ring*.

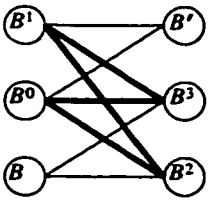


Figure 3.7: A 2 process q^*

Definition 3.6 *A two process* quad ring *containing the blocks* B^0, B^1, B^2 *and* B^3 *where* $B^0 \rightarrow B^1$ *and* $B^2 \rightarrow B^3$ *is a* q^* *if and only if there exist blocks* B *and* B' *such that the following relationships hold:*

- $B \rightarrow B'$
- $B \rightarrow B^0$, $B \| B^2$ *and* $B \| B^3$
- $B^3 \rightarrow B'$, $B' \| B^0$ *and* $B' \| B^1$

Any execution that contains a two process q^* as a subgraph of the concurrency graph cannot be accurately represented as a concurrency map. Lemma 3.2 uses the definition above and the construction of the concurrency map to formulate a proof.

Lemma 3.2 *A two process q^* cannot be accurately represented as a concurrency map.*

Proof: Assume we are given a six node subgraph of $\bar{\mathbf{H}}$ as defined in definition 3.6 and shown in figure 3.7.

$$\lfloor B^1 \rfloor = \lceil B^2 \rceil \qquad B^1 \| B^2,\ B^0 \| B^3 \text{ and lemma 3.1} \qquad (3.38)$$

$$\lfloor B^1 \rfloor = \lceil B^3 \rceil \qquad B^1 \| B^3,\ B^0 \| B' \text{ and lemma 3.1} \qquad (3.39)$$

$$\lceil B^2 \rceil = \lceil B^3 \rceil \qquad (3.38) \text{ and } (3.39) \qquad (3.40)$$

$$\lfloor B^3 \rfloor \leq \lceil B^3 \rceil \qquad \text{definitions 3.2 and 3.3} \qquad (3.41)$$

$$\lceil B^2 \rceil \geq \lfloor B^3 \rfloor \qquad (3.40) \text{ and } (3.41) \qquad (3.42)$$

$$\lceil B^2 \rceil \leq \lfloor B^3 \rfloor \qquad B^2 \rightarrow B^3 \text{ and definition 3.4 (2)} \qquad (3.43)$$

$$\lceil B^2 \rceil = \lfloor B^3 \rfloor \qquad (3.42) \text{ and } (3.43) \qquad (3.44)$$

$$\lfloor B^3 \rfloor = \lceil B \rceil \qquad B^3 \| B,\ B^2 \| B^0 \text{ and lemma 3.1} \qquad (3.45)$$

$$\lceil B \rceil = \lceil B^2 \rceil \qquad (3.44) \text{ and } (3.45) \qquad (3.46)$$

$$\lceil B \rceil = \lfloor B^1 \rfloor \qquad (3.38) \text{ and } (3.46) \qquad (3.47)$$

$$\lfloor B' \rfloor = \lceil B^0 \rceil \qquad B' \| B^0,\ B^3 \| B^1 \text{ and lemma 3.1} \qquad (3.48)$$

$$\lceil B^0 \rceil \leq \lfloor B^1 \rfloor \qquad B^0 \rightarrow B^1 \text{ and definition 3.4 (2)} \qquad (3.49)$$

$$\lfloor B' \rfloor \leq \lfloor B^1 \rfloor \qquad (3.48) \text{ and } (3.49) \qquad (3.50)$$

$$\lfloor B' \rfloor \leq \lceil B \rceil \qquad (3.47) \text{ and } (3.50) \qquad (3.51)$$

$$\lfloor B' \rfloor > \lceil B \rceil \qquad B' \rightarrow B \text{ and definition 3.4 (3)} \qquad (3.52)$$

$$\textit{False} \qquad (3.51) \text{ and } (3.52) \qquad (3.53)$$

■

A *quad ring* that contains blocks from three processes has the row division needed to produce a q^*. Consider the execution of figure 3.6 again. Blocks B_i^0, B_j^0, B_k^0 and B_k^1 form a *quad ring*. A message is transmitted from P_j to P_i creating an inter-process causal relationship. This relationship is displayed by separating the send and receive blocks into different rows of the concurrency map. The blocks in P_k are concurrent to both the send and receive blocks and temporally related to each other. Note that the causal link could be transitive instead of direct. We only require that the link exists. All three process *quad rings* are also q^*s. Lemma 3.3 proves the impossibility of representing a three process q^* as a concurrency map. Figure 3.8 (a) shows a simplified three process concurrency map that will be used in the proof.

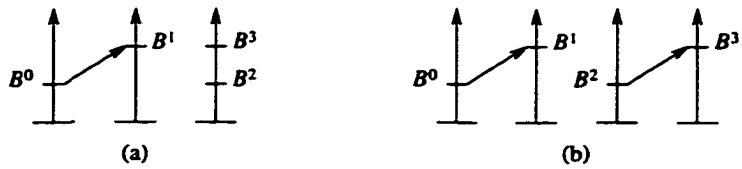


Figure 3.8: A 3 process q^* and a 4 process q^*

Lemma 3.3 *A three process q^* cannot be accurately represented as a concurrency map.*

Proof: We proceed by contradiction. Given a three process q^* such that shown in figure 3.8 (a) where $B^0 \rightarrow B^1$, $B^2 \rightarrow B^3$, and $\mathcal{P}(B^0) \neq \mathcal{P}(B^1)$, assume that an accurate

concurrency map can be constructed.

$\lfloor B^1 \rfloor \leq \lceil B^2 \rceil$	$B^1 \| B^2$ and lemma 3.1	(3.54)
$\lfloor B^3 \rfloor \leq \lceil B^0 \rceil$	$B^3 \| B^0$ and lemma 3.1	(3.55)
$\lceil B^2 \rceil \leq \lfloor B^3 \rfloor$	$B^2 \rightarrow B^3$ and definition 3.4 (2)	(3.56)
$\lfloor B^1 \rfloor \leq \lfloor B^3 \rfloor$	(3.54) and (3.56)	(3.57)
$\lfloor B^1 \rfloor \leq \lceil B^0 \rceil$	(3.55) and (3.57)	(3.58)
$\lceil B^0 \rceil < \lfloor B^1 \rfloor$	$B^0 \rightarrow B^1$ and definition 3.4 (3)	(3.59)
False	(3.58) and (3.59)	(3.60)

■

The necessary row division in the concurrency map is also present when the blocks of the *quad ring* are from four processes. Blocks B_i^0, B_j^0, B_k^0 and B_l^0 of figure 3.6 form a four process *quad ring*. Processes P_j and P_k concurrently send messages to P_i and P_l which concurrently receive those messages. Note that this is not the only execution that could produce a *quad ring*. The causal relationships between processes could be transitive instead of direct. We only require that the causal relationships exist. Just as in the three process case, all four process *quad rings* are considered q^*s. Figure 3.8 (b) shows a simplified time-space diagram of a four process q^* and lemma 3.4 proves the impossibility of accurately displaying the q^* as a concurrency map. Note the similarity between the proofs of lemmas 3.3 and 3.4. The three-process case attributes one causal link to temporal order while the four-process case attributes both causal links to communication. The sources of the links accounts for the only difference between the proofs.

Lemma 3.4 *A four process q^* cannot be accurately represented as a concurrency map.*

Proof: We proceed by contradiction. Given a four process q^* such that $B^0 \rightarrow B^1$ and $B^2 \rightarrow B^3$ as shown in figure 3.8 (b), assume that an accurate concurrency map can be constructed.

$\lfloor B^1 \rfloor \leq \lceil B^2 \rceil$	$B^1 \| B^2$ and lemma 3.1	(3.61)
$\lfloor B^3 \rfloor \leq \lceil B^0 \rceil$	$B^3 \| B^0$ and lemma 3.1	(3.62)
$\lceil B^2 \rceil < \lfloor B^3 \rfloor$	$B^2 \rightarrow B^3$ and definition 3.4 (3)	(3.63)
$\lfloor B^1 \rfloor < \lfloor B^3 \rfloor$	(3.61) and (3.63)	(3.64)
$\lfloor B^1 \rfloor < \lceil B^0 \rceil$	(3.62) and (3.64)	(3.65)
$\lceil B^0 \rceil < \lfloor B^1 \rfloor$	$B^0 \rightarrow B^1$ and definition 3.4 (3)	(3.66)
False	(3.65) and (3.66)	(3.67)

■

We have considered q^* graphs for systems of two, three and four processes. As stated earlier, since the graph is based on concurrency, there are no such graphs for a single process system. The events of a single process are totally ordered and are not concurrent. Lemma 3.5 proves that there are no q^* graphs for systems of five or more processes.

Lemma 3.5 *$\nexists$ k-process q^* where $k > 4$.*

Proof: A q^* is defined as a *quad ring* with an additional constraint. Namely, that an interprocess causal link is present. By proposition 2.3 we know that the *quad ring* contains no more than four events. Therefore, the q^* can contain no more than four events. If each

event is from a different process, there are only four processes involved. Hence, there are no k-process q^*s where k is greater than four. ■

The next section lends credence to the importance of q^*. We show that not only is the construction of an accurate representation by a concurrency map impossible if the q^* subgraph of $\bar{\mathbf{H}}$ is present, but it is guaranteed if a q^* subgraph is not present.

3.3 q^* and Concurrency Maps

If we are given an execution that contains any q^* as a subgraph of the concurrency graph, then an accurate representation as a concurrency map is not possible. Since there are only three cases in which a q^* is defined, and each has been proven, the proof of the theorem is straightforward.

Theorem 3.3 *If there exists a q^* subgraph of $\bar{\mathbf{H}}$ then an accurate concurrency map representation of $\mathbf{H}$ cannot be constructed.*

Proof: Follows directly from lemmas 3.2, 3.3 and 3.4. ■

As stated in the previous section, this is not the limit of the influence of the *quad ring*. We now show that an accurate representation as a concurrency map is guaranteed if the q^* subgraph is not present. We proceed by induction on the number of blocks in the concurrency map. As a base case we show that all four-node subgraphs that are not a q^* can be represented as a concurrency map. Then we show that a construction technique can be used to add nodes to an accurate concurrency map producing an accurate representation of a $k+1$ node graph for all values of k.

Lemma 3.6 *All induced four-node subgraphs, $\bar{h} \subseteq \bar{\mathbf{H}} : \bar{h} \neq q^*$ can be accurately represented as a concurrency map.*

Proof: We proceed by exhaustive enumeration. There are only two circumstances under which an induced four-node subgraph, $\bar{h} \subseteq \bar{\mathbf{H}}$, is not a q^*. Either $\bar{h}$ is a two-process *quad ring* where the additional constraints do not hold or $\bar{h}$ is not a *quad ring*. Construction of a correct representation of a two process *quad ring* which is not a q^* by a concurrency map is trivial. As indicated previously, a single row in a two column concurrency map is sufficient to display this situation. We now consider each possible alternate configuration of $\bar{h}$ that are not a *quad ring*. We will consider all possible arrangements of arcs between four nodes.

Let us first consider the cases where either all four nodes are fully connected, or where none of the nodes are connected. Figure 3.9 shows the only two possible configurations that these two situations can produce. These two graphs are numbered 1 and 10, respectively, in the final list of figure 3.18.

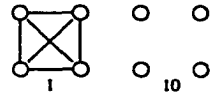


Figure 3.9: Configurations of $\bar{h}$ with 6 or 0 arcs

If five arcs are present of the possible six, we must consider $\binom{6}{5} = 6$ different graphs. Each one shown in figure 3.10 has a different arc removed. It is easy to see that graphs 1 and 2 are isomorphic. Likewise, graphs 3, 4, 5 and 6 are isomorphic. In figure 3.11 we show a labeling of graphs 1 and 5 indicating the isomorphism of the graphs. Graph 2 of figure 3.18 represents the only possible configuration of four nodes with five arcs.

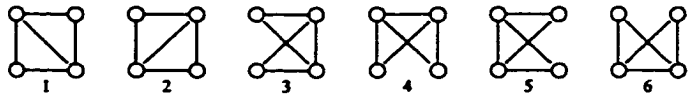


Figure 3.10: Configurations of $\bar{h}$ with 5 arcs

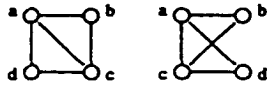


Figure 3.11: Equivalence of 5 arc nodes

If four arcs are present, there are $\binom{6}{4} = 15$ different graphs we must consider. These graphs are shown in figure 3.12. In this case, graphs 13, 14 and 15 represent forbidden *quad ring*'s and are not considered. Of the remaining graphs, numbers 1 through 8 are rotations and inversions of each other. They are therefore isomorphic. Graphs numbered 9 through 12 are also rotations and inversions of each other, and are isomorphic. From these two remaining graphs we show numbers 1 and 9 in figure 3.13. The labels attached to the nodes indicate the rearrangement that will produce one from the other. These are also the same graph which is number 3 in figure 3.18.

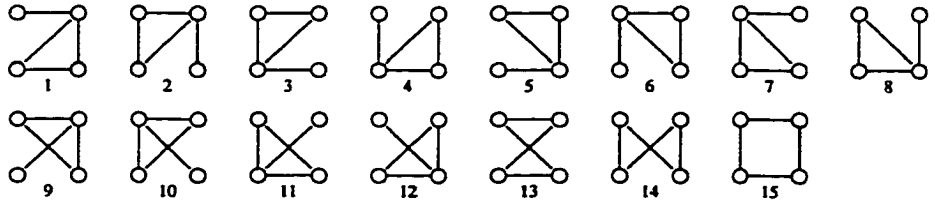


Figure 3.12: Configurations of $\bar{h}$ with 4 arcs

If only three arcs are present in the graphs, there are $\binom{6}{3} = 20$ different graphs we must consider. These graphs are shown in figure 3.14. The graphs can be divided into five groups

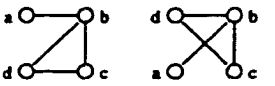


Figure 3.13: Equivalence of 4 arc nodes

of four each. Graphs 17 through 20 are in one group because they are rotations. Graphs 13 through 16 are in another group since they are also rotations. The remaining twelve graphs have representatives in figure 3.15. Numbers 1 through 4 are represented by the graph on the left, graphs 5 through 8 are represented by the graph in the center, and graphs 9 through 12 are represented by the graph on the right. Each representative graph is labeled to show that they are isomorphic to the same underlying graph. Therefore, we have three different graphs with three arcs which are numbered 4, 5 and 6 in figure 3.18.

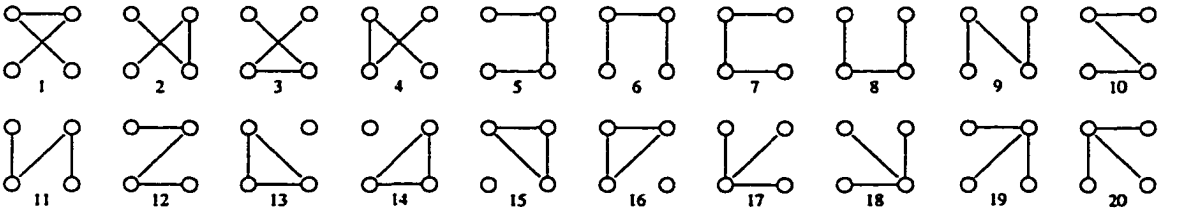


Figure 3.14: Configurations of $\bar{h}$ with 3 arcs

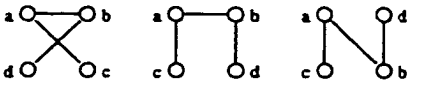


Figure 3.15: Equivalence of 3 arc nodes

We next consider the graphs with only two arcs where we have $\binom{6}{2} = 15$ possibilities. These possibilities are shown in figure 3.16. Note that the arcs either share a common node or are disjoint. Graph number 7 in figure 3.18 shows two disjoint arcs while graph number

8 shows two arcs that share a common node. All the two arc graphs in figure 3.16 are isomorphic to one of the two arc graphs in figure 3.18.

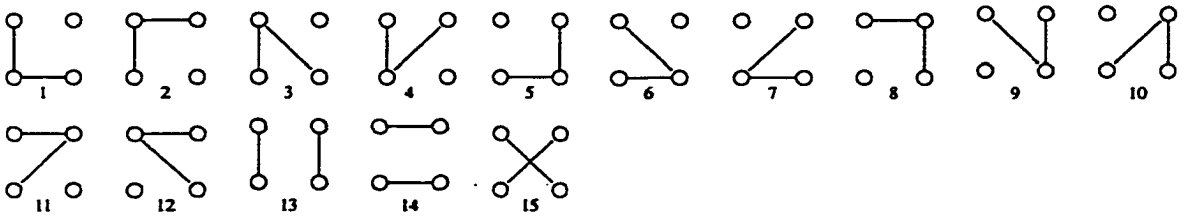


Figure 3.16: Configurations of $\bar{h}$ with 2 arcs

There are $\binom{6}{1} = 6$ graphs we must consider with a single arc. These are shown in figure 3.17. By rearranging the nodes we can easily transform one of the graphs into another. The single arc graphs are isomorphic to number 9 in figure 3.18.

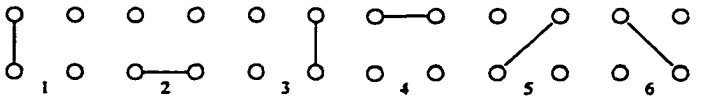


Figure 3.17: Configurations of $\bar{h}$ with 1 arc

We have considered all possibilities of undirected arcs in a four node graph. The order of the nodes will become important when we consider the graphs h from which each $\bar{h}$ could have been derived. However, the position of a node in $\bar{h}$ is irrelevant. We have rearranged the position of the nodes to show that all possibilities are represented by a set of ten graphs. These are shown in figure 3.18.

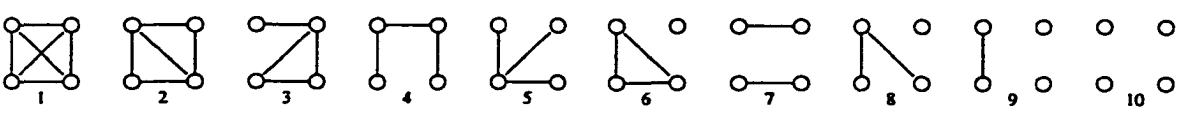


Figure 3.18: Possible non-*quad ring* configurations of $\bar{h}$

Given that the absence of an undirected edge B_1B_2 in $\bar{h}$ implies a directed edge, either B_1B_2 or B_2B_1, in h, we can enumerate the possible configurations of h. Figure 3.19 shows that there are 41 possible directed graphs that have one of the $\bar{h}$ graphs as an undirected complement. To clarify the derivations of the directed complements of each $\bar{h}$, we present table 3.1. Each of the numbered $\bar{h}$ graphs is listed to the left while the right portion of that table shows which of the 41 directed complements were derived from it.

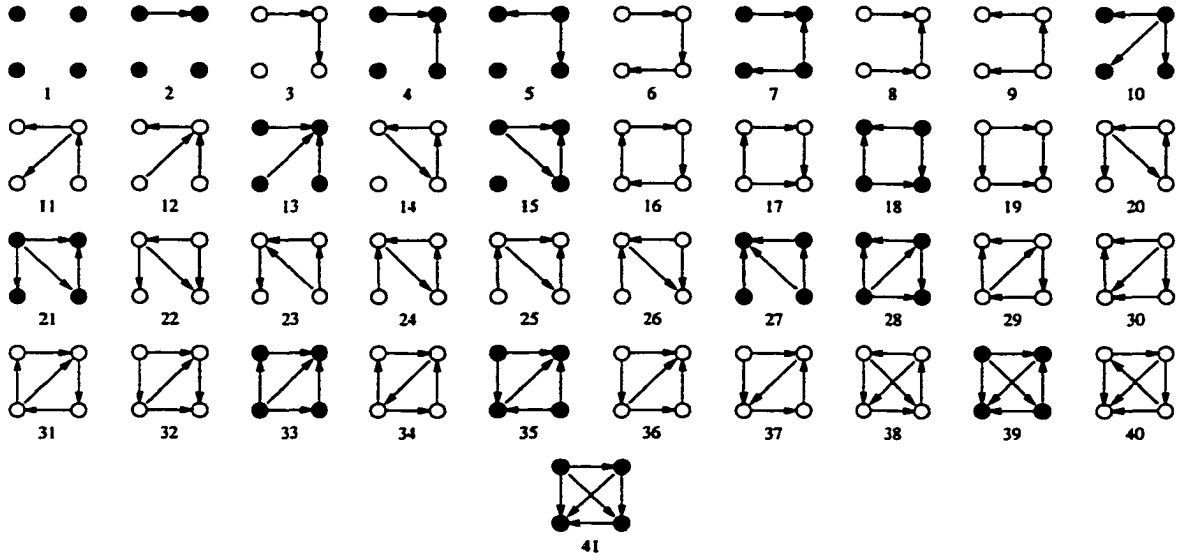


Figure 3.19: Directed graph derivations of $\bar{h}$

Some of the directed graphs of figure 3.19 represent impossible executions of a distributed system. For example consider the directed graph numbered 3 in the figure. From proposition 2.1 and the transitivity of the causal relation we know that $B \rightarrow B' \rightarrow B'' \Rightarrow B \rightarrow B''$. However, the graph indicates that $B \rightarrow B' \rightarrow B''$ but $B \| B''$. In other words, an arc from the top left node to the bottom right node should be present, but is not. Therefore the graph does not represent a possible execution of a distributed system. The cycle present in

$\bar{h}$	derived digraphs
1	1
2	2
3	3 4 5
4	6 7 8 9
5	10 11 12 13
6	14 15
7	16 17 18 19
8	20 21 22 23 24 25 26 27
9	28 29 30 31 32 33 34 35 36 37
10	38 39 40 41

Table 3.1: Directed graphs derived from each $\bar{h}$

directed graph number 14 disqualifies it as well.

Only nodes with solid vertices do not violate either proposition 2.1 or proposition 2.2. For each of the possible directed graph representations, it is impossible to determine whether the causal links are due to interprocess communication or to intraprocess temporal ordering. For this reason, multiple concurrency maps are possible for a single directed graph. Figure 3.20 shows the 70 possible concurrency maps derived from the valid directed graphs of figure 3.19.

Again, we supply a cross reference that indicates which concurrency maps were derived from the directed graphs. Notice that some of the concurrency maps have multiple entries in table 3.2. This indicates the difficulty in deducing the number of processes involved in the graph. For example, consider the last graph of figure 3.19. Since all nodes of the graph are causally related, any number of processes from one to four inclusive could be represented.

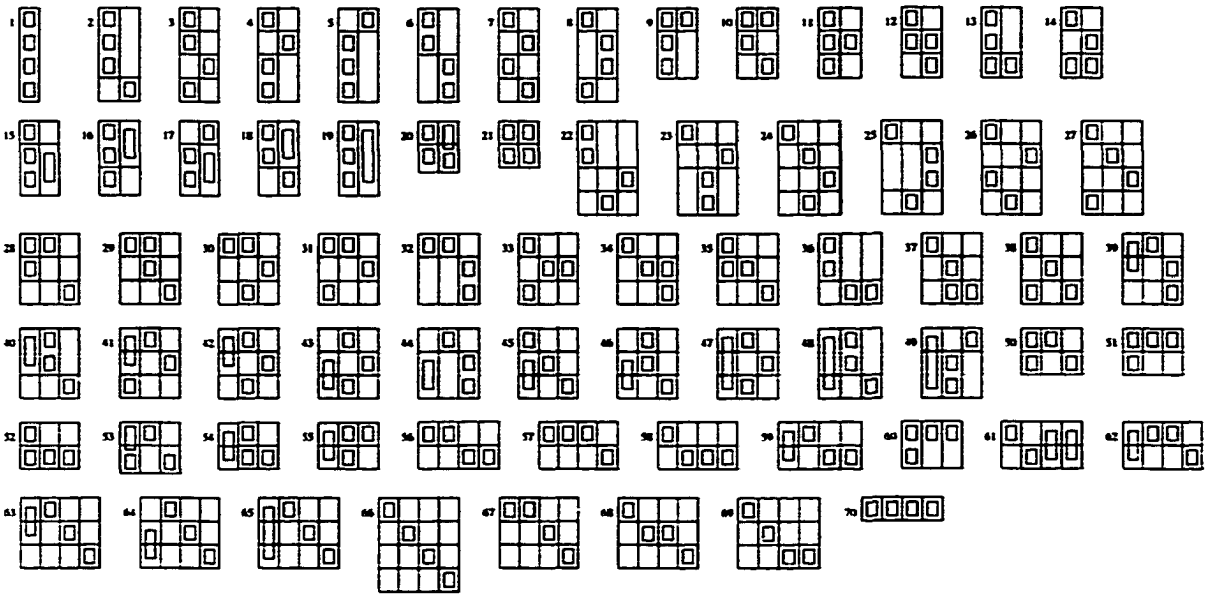


Figure 3.20: Directed graph derivations of $\bar{h}$

By exhaustive enumeration we have established that any four-node subgraph of $\bar{\mathbf{H}}$ which is not isomorphic to q^* can be accurately represented as a concurrency map. ■

3.3.1 Partitions of h

Assume we are given a k-node induced subgraph, $\bar{h} \subseteq \bar{\mathbf{H}}$, such that $q^* \not\subseteq \bar{h}$ and a node $n \in \bar{\mathbf{H}}$ such that $n \notin \bar{h}$. Let $\bar{h}'$ be an induced subgraph of $\bar{\mathbf{H}}$ with the nodes of $\bar{h}$ and the node n, such that $q^* \not\subseteq \bar{h}'$. We can define three partitions of the nodes of $\bar{h}$ based on the relationship of each of these nodes to n.

digraphs	derivative concurrency maps
1	70
2	61 60
4	55 62
5	55 62
7	20 53 54 59
10	52 58
13	51 57
15	19 47 48 49 65
18	21 50 56
21	15 17 43 44 45 46 64
27	16 18 39 40 41 42 63
28	13 14 36 37 38 69
33	11 12 33 34 35 68
35	9 10 28 31 32 67
39	1 2 3 4 5 6 7 8 22 23 24 25 26 27 66
41	1 2 3 4 5 6 7 8 22 23 24 25 26 27 66

Table 3.2: Concurrency maps derived from each directed graph

Definition 3.7 $A = \{B : B \in \bar{h} \wedge n \rightarrow B\}$.

Definition 3.8 $B = \{B : B \in \bar{h} \wedge B \rightarrow n\}$.

Definition 3.9 $C = \{B : B \in \bar{h} \wedge B \| n\}$.

The three sets are *after*, *before*, and *concurrent*. They contain the nodes of $\bar{h}$ that causally succeed, causally precede, and are concurrent to node n, respectively. From definitions 2.4 and 2.7 we know that all nodes of $\bar{\mathbf{H}}$ are either causally or concurrently related. Therefore A, B, and C represent a valid partition of any induced subgraph of $\bar{\mathbf{H}}$. We also

know the following properties hold. No proofs are given since the derivation is directly from the definitions of causality and concurrency.

Property 3.1 $\forall a \in A, \forall b \in B, b \rightarrow a$.

Property 3.2 $\forall a \in A, \forall c \in C, a \not\rightarrow c$.

Property 3.3 $\forall b \in B, \forall c \in C, c \not\rightarrow b$.

Since we are assured that no *quad rings* are present in either $\bar{h}$ or $\bar{h}'$, we can deduce several impossibilities for the concurrency map with $k+1$ nodes. In lemma 3.7 we show that under some conditions we can be certain of the processes in which blocks are executed. We are given two causally related blocks from partition C and a single block from partition B. If the later block of C is concurrent to the block from B, then we are assured that both blocks from C are executed by the same process and the block from B is executed by the same process as n. If either condition was not met, a forbidden q^* would be formed.

Lemma 3.7 $\forall c, c' \in C \, \forall b \in B : c' \rightarrow c \wedge c \| b \Rightarrow \mathcal{P}(c) = \mathcal{P}(c') \wedge \mathcal{P}(b) = \mathcal{P}(n)$

Proof: Assume we are given $c, c' \in C$ and $b \in B$ such that $c' \rightarrow c$ and $b \| c$.

$c' \not\rightarrow b$	property 3.3	(3.68)
$b \not\rightarrow c'$	$c' \rightarrow c$, $c \| b$ and property 2.5	(3.69)
$c' \| b$	(3.68) and (3.69)	(3.70)

If either $\mathcal{P}(c) \neq \mathcal{P}(c')$ or $\mathcal{P}(b) \neq \mathcal{P}(n)$, then the set $\{c, c', b, n\} \subseteq \bar{h}'$ is a three process q^*. Consider figure 3.21 where concurrent relationships are shown as solid, undirected lines,

and causal relationships are shown as dashed arrows. If exactly one of the causal links is interprocess, then a three process q^* is formed. If both $\mathcal{P}(c) \neq \mathcal{P}(c')$ and $\mathcal{P}(b) \neq \mathcal{P}(n)$ then the set $\{c, c', b, n\} \subseteq \bar{h}'$ is a four process q^*. Referring again to figure 3.21, a four process q^* is formed when both causal links are interprocess. Either case contradicts the assumption that $q^* \not\subseteq \bar{h}'$. Therefore, $\mathcal{P}(c) = \mathcal{P}(c') \wedge \mathcal{P}(b) = \mathcal{P}(n)$. ■

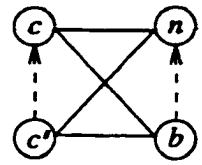


Figure 3.21: Block arrangements forming a q^* with a block from B

In lemma 3.8 we assume only that there exist two causally related blocks in C. By definition, we know that these blocks are concurrent to n. That provides sufficient leverage to restrict the existence of blocks of B. Since a block from B that is concurrent to c (and therefore concurrent to c') would form a *quad ring* if either c and c' were executed by different processes or the block from B and n were executed by different processes. If either $\lceil b \rceil \geq \lfloor c \rfloor$ or $\mathcal{P}(b) \neq \mathcal{P}(n)$, a forbidden q^* is formed.

Lemma 3.8 $\forall c, c' \in C : c' \rightarrow c \Rightarrow \not\exists b \in B : \lceil b \rceil \geq \lfloor c \rfloor \wedge \mathcal{P}(b) \neq \mathcal{P}(n)$

Proof: We proceed by contradiction. Assume there exists $b \in B$ such that $\lceil b \rceil \geq \lfloor c \rfloor$ and $\mathcal{P}(b) \neq \mathcal{P}(n)$.

$$b \not\rightarrow c \qquad \lceil b \rceil \geq \lfloor c \rfloor \text{ and definition 3.4} \qquad (3.71)$$

$$c \not\rightarrow b \qquad \text{property 3.3} \qquad (3.72)$$

$b\|\|c$	(3.71), (3.72) and definition 2.7	(3.73)
$\mathcal{P}(b) = \mathcal{P}(n)$	$c' \rightarrow c$, (3.73) and lemma 3.7	(3.74)
False	$\mathcal{P}(b) \neq \mathcal{P}(n)$ and (3.74)	(3.75)

■

Where the previous two corollaries show the necessary configuration of causally related blocks from C and a block from B, the following two corollaries show similar properties as they related to blocks from A. Lemma 3.9 again assumes that we are given two causally related blocks from C. Also assumed is a block from A that is concurrent to the preceding block from C. The proof shows that the processes executing all three blocks are defined relative to each other. Specifically, both blocks from C are executed in the same process and the block from A is executed in the same process as n.

Lemma 3.9 $\forall c, c' \in C\, \forall a \in A : c \rightarrow c' \wedge c\|a \Rightarrow \mathcal{P}(c) = \mathcal{P}(c') \wedge \mathcal{P}(a) = \mathcal{P}(n)$

Proof: Assume there exists $c, c' \in C$ and $a \in A$ such that $c \rightarrow c'$ and $c\|a$.

$a \not\rightarrow c'$	property 3.2	(3.76)
$c' \not\rightarrow a$	assumption and property 2.5	(3.77)
$c'\|\|a$	(3.76) and (3.77)	(3.78)

If either $\mathcal{P}(c) \neq \mathcal{P}(c')$ or $\mathcal{P}(a) \neq \mathcal{P}(n)$, then the set $\{c, c', a, n\} \subseteq \bar{h}'$ is a three process q^*. Consider figure 3.22 where concurrent relationships are shown as solid, undirected lines, and causal relationships are shown as dashed arrows. If exactly one of the causal links is

interprocess, then a three process q^* is formed. If $\mathcal{P}(c) \neq \mathcal{P}(c')$ and $\mathcal{P}(a) \neq \mathcal{P}(n)$ then the set is a four process q^*. Referring again to figure 3.22, a four process q^* is formed when both causal links are interprocess. Either case contradicts the assumption that $q^* \not\subseteq \bar{h}'$. Therefore, $\mathcal{P}(c) = \mathcal{P}(c')$ and $\mathcal{P}(a) = \mathcal{P}(n)$. ■

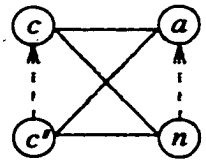


Figure 3.22: Block arrangements forming a q^* with a block from A

To avoid the creation of a *quad ring* in the concurrency map given a pair of causally related blocks of C, we restrict the placement of blocks from A in much the same way we did for blocks of B. Blocks from A which are concurrent to c (and therefore concurrent to c') must be executed by the same process as n. Otherwise, $\{c, c', a, n\}$ would form a *quad ring*. If a block from A is concurrent to both c and c', we are also assured by the previous lemma that the two blocks from C are executed in the same process.

Lemma 3.10 $\forall c, c' \in C : c \rightarrow c' \Rightarrow \nexists a \in A : \lfloor a \rfloor \leq \lceil c \rceil \wedge \mathcal{P}(a) \neq \mathcal{P}(n)$

Proof: We proceed by contradiction. Assume there exists $a \in A$ such that $\lfloor a \rfloor \leq \lceil c \rceil$ and $\mathcal{P}(a) \neq \mathcal{P}(n)$.

$c \not\rightarrow a$	assumption and corollary 3.1	(3.79)
$a \not\rightarrow c$	property 3.2	(3.80)
$c \| a$	(3.79), (3.80) and definition 2.7	(3.81)

$\mathcal{P}(a) = \mathcal{P}(n)$	premise, (3.81) and lemma 3.9	(3.82)
False	assumption and (3.82)	(3.83)

■

Let the two values, α and β represent the limits of the A and B sets. The smallest numbered row of the concurrency map in which a block from set A appears is labeled α and the highest numbered row in which a block from set B appears is labeled β. Figure 3.23 shows six blocks from B, four from A, and three from C. Also shown are the locations of β and α in the concurrency map. It is possible that either A, B or both may be empty, in which case special care must be taken in the definitions of the set limits. The following definitions insure that $\alpha > \beta$.

Definition 3.10 *Given $\bar{h} \subseteq \bar{\mathbf{H}}$ such that $\bar{h} \neq q^*$, the partitions A, B, and C, and the accurate concurrency map representation of $\bar{h}$, we define β as follows.*

$$\beta = \begin{cases} \max(\lceil b \rceil \forall b \in B) & \text{if } B \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

Definition 3.11 *Given $\bar{h} \subseteq \bar{\mathbf{H}}$ such that $\bar{h} \neq q^*$, the partitions A, B, and C, and the accurate concurrency map representation of $\bar{h}$, we define α as follows.*

$$\alpha = \begin{cases} \min(\lfloor a \rfloor \forall a \in A) & \text{if } A \neq \emptyset \\ \max(\lceil v \rceil \forall v \in \bar{h}, \beta + 1) & \text{otherwise} \end{cases}$$

Let Ψ be an accurate concurrency map representation of the relations in $\bar{h}$. From Ψ we construct $\widehat{\Psi}$, a concurrency map that accurately depicts the relations of $\bar{h}'$. The construction is based on the images of blocks in Ψ. Both α and β are used extensively in

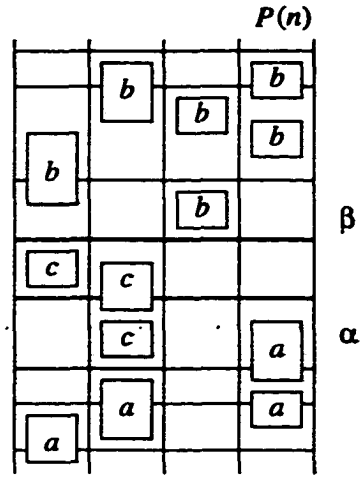


Figure 3.23: β and α with sets A, B, and C

the transformation. The next section defines the transformation of Ψ into $\widehat{\Psi}$ by altering the images of each block in Ψ and placing the additional block n.

3.4 Transformation of the concurrency map

We now transform the known correct concurrency map, Ψ, into $\widehat{\Psi}$, the concurrency map correctly displaying the relationships of Ψ with the addition of n. The image of each block of Ψ is used to calculate the image of the corresponding block in $\widehat{\Psi}$ to preserve the relationships in Ψ. Although not explicitly stated in the construction, we assume that the temporal order of nodes of Ψ is preserved. That is, multiple nodes enclosed within a single row cannot be reordered in the construction. Two nodes of a process completely enclosed in a single row would maintain a correct image relationship even if they were temporally reversed in the associated concurrency map. Only the relative order of the reordered blocks would be incorrect.

We begin by placing n in $\widehat{\Psi}$ in rows that will ultimately be between the blocks of B and A. By using the values of β and α, we take into account all blocks from both B and A. The image of n in $\widehat{\Psi}$, $\mathcal{I}(n)$, is from row $\beta+1$ to $\alpha+1$.

$$\mathcal{I}(n) = \{\beta+1 \ \ldots \ \alpha+1\} \tag{3.84}$$

Later transformation equations will show that we shift the blocks of A down by two rows. By leaving the blocks of B placed before $\beta+1$ and moving the blocks of A to after $\alpha+1$, we free rows $\beta+1$ to $\alpha+1$ for the placement of n.

In the next section we will show that this assignment insures the accurate display of the causal relationships among all blocks of B and A with n. Elements of C are not taken into account expressly since they will ultimately be shown concurrent to n. Their individual transformations will insure that part of their images fall in the range $(\beta+1 \ldots \alpha+1)$. All blocks from process P_i are placed in column i in both Ψ and $\widehat{\Psi}$.

3.4.1 Transforming the blocks of partition B

The blocks of partition B retain their original positions in most cases. However, some situations force blocks to be moved. We will examine the transformation of the minimum and maximum values of the image of blocks from B in turn. The transformation equations (3.85) and (3.86) collectively identify where they should be placed in $\widehat{\Psi}$.

We begin by examining the maximum of the image of each block. We know that n will be placed beginning at row $\beta+1$, and all blocks from other processes that causally precede n must be found higher in the map, i.e., $\lceil \hat{b} \rceil < \beta+1$. From the definition of β we are

assured that the images in Ψ of the blocks of B meet this criterion. Hence, no alteration is needed and $\lceil \hat{b} \rceil = \lceil b \rceil$. However, if for some b, $\mathcal{P}(b) = \mathcal{P}(n)$, then we must consider the possible relationships with both the other blocks of B and the blocks of C.

It is possible that some $c \in C$, concurrent to both b and n, is placed such that $\lfloor \hat{c} \rfloor = \beta + 1$. In which case, to show the concurrency between b and c, we must let $\lceil \hat{b} \rceil = \beta + 1$ as well. Under most conditions the images of all $b \in B$ where $\mathcal{P}(b) = \mathcal{P}(n)$ should be extended into the $\beta + 1$ row. The following paragraphs identify the circumstances under which the extension should not be performed. Equation 3.85 defines the transformation.

$$\lceil \hat{b} \rceil = \begin{cases} \beta + 1 & \text{if } [\mathcal{P}(b) = \mathcal{P}(n)] \\ & \wedge\, [\nexists b', b'' \in B : \mathcal{P}(b') = \mathcal{P}(b) \neq \mathcal{P}(b'') \wedge\ b \rightarrow b' \wedge \lceil b'' \rceil \geq \lfloor b' \rfloor] \\ & \wedge\, [\nexists v \in B \cup C : \mathcal{P}(v) \neq \mathcal{P}(b) \wedge\ b \rightarrow v \wedge \lfloor v \rfloor \leq \beta] \\ \lceil b \rceil & \text{otherwise} \end{cases} \tag{3.85}$$

Consider the situation shown in figure 3.24(a). The block b is executed by the same process as n. If we were to extend the maximum of the image of b into row $\beta + 1$, we force b' into an incorrect position. Since blocks are not allowed to overlap, we must place b' such that $\lfloor \hat{b'} \rfloor$ is also $\beta + 1$. Since b'' and n are executed by different processes, the placement of b'' will not change, and the concurrency between $\hat{b'}$ and $\hat{b''}$ will not be correctly shown. Therefore, the image of b is not altered in this situation.

The second clause, illustrated in figure 3.24(b), considers blocks from both partitions B and C. If there exists a block from another process that causally follows b situated in Ψ at least partially above β, then the image of $\hat{b}$ cannot be stretched into the $\beta + 1$ row. The reason for retaining the original position in this case is that the other block, either in B or C, will not be moved below $\beta + 1$. If the block is in B, it will retain its original position

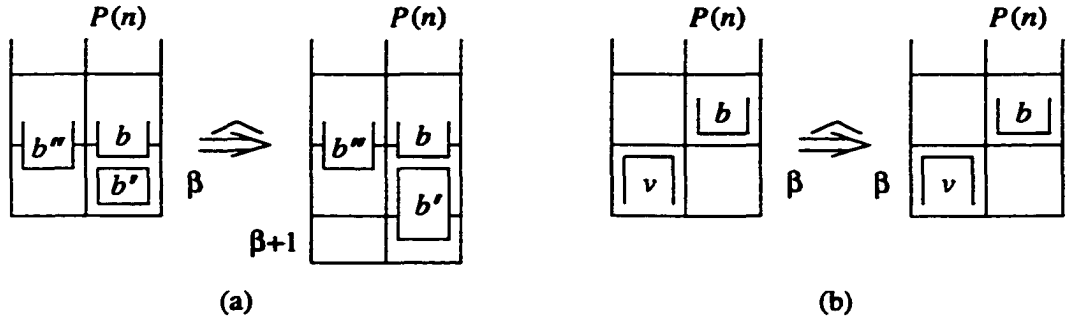


Figure 3.24: Special cases in the transformation of $b \in B$

since it is executed in a process other than $\mathcal{P}(n)$. A block in C meeting these criteria will never be moved lower than row $\beta + 1$. If b was allowed to stretch into $\beta + 1$, it would incorrectly be shown concurrent to the block that causally follows it.

The minimum of the image of each block of B is transformed in a manner similar to the transformation of the maximum. In most cases, no alteration is needed to correctly display the relationships with other blocks of the system. However, if the block in question is in the same process as n then we must consider other possible scenarios. Referring again to figure 3.24(a), consider the transformation of b'. If we ignore b'', we are unsure whether the maximum of b should be stretched into row $\beta + 1$. If it is, then to avoid overlapping images, we must also move the minimum of b' into $\beta + 1$. The movement to $\beta + 1$ requires that another block must temporally precede b' as b does in the figure. It also requires that a block with concurrent relationships to b and b' such as b'' in the figure does not exist in Ψ. If either requirement is not met, the minimum of b' must remain in its original position. If it was allowed to be moved into $\beta + 1$, then it would no longer be shown concurrent to b''.

On the other hand, if the temporally preceding block (b) is found in Ψ and the concurrent block (b'') is not found, then the movement can take place. Both the minimum and maximum of the image of b' as well as the maximum of the image of b are moved into row $\beta+1$. The minimum of the image of b is considered in a similar manner and may be moved into $\beta+1$ as well. Equation 3.86 formalizes the transformation.

$$\lfloor \hat{b} \rfloor = \begin{cases} \beta+1 & \text{if } [\mathcal{P}(b) = \mathcal{P}(n)] \\ & \wedge\, [\exists b' \in B : \mathcal{P}(b') = \mathcal{P}(b) \wedge b' \rightarrow b] \\ & \wedge\, [\nexists b'' \in B : \mathcal{P}(b'') \neq \mathcal{P}(b) \wedge \lceil b'' \rceil \geq \lfloor b \rfloor] \\ \lfloor b \rfloor & \text{otherwise} \end{cases} \tag{3.86}$$

Using these equations we prove that the causal relationships among all blocks in B are maintained through the transformation. We begin by showing that the causal relationships in Ψ are preserved in the transformation. The proof is based on the definitions of β and B as well as the definition of a concurrency map. In the first case we assume that both blocks of B are executed by the same process. This allows us to ascertain their transformed positions relative to their original positions. In the second and third cases, we assume that b and b' were executed by different process. Case two assumes that b was executed by the same process as n while case three assumes the opposite.

Lemma 3.11 $\forall b, b' \in B : b \rightarrow b' \Rightarrow \hat{b} \rightarrow \hat{b'}$.

Proof:

CASE 1: $\mathcal{P}(b) = \mathcal{P}(b')$

$$\lceil b \rceil \leq \lfloor b' \rfloor \qquad b \rightarrow b',\ \mathcal{P}(b) = \mathcal{P}(b') \text{ and definition 3.4 (2)} \tag{3.87}$$

$\lceil \hat{b} \rceil = \lceil b \rceil$	$b \rightarrow b'$ and (3.85)	(3.88)
$\lfloor b' \rfloor \leq \beta$	definition 3.10	(3.89)
$\lfloor \hat{b'} \rfloor = \lfloor b' \rfloor \vee \lfloor \hat{b'} \rfloor = \beta + 1$	(3.86)	(3.90)
$\lfloor \hat{b'} \rfloor \geq \lfloor b' \rfloor$	(3.89) and (3.90)	(3.91)
$\lceil \hat{b} \rceil \leq \lfloor \hat{b'} \rfloor$	(3.87), (3.88) and (3.91)	(3.92)
$\hat{b} \rightarrow \hat{b'}$	(3.92) and definition 3.4 (2)	(3.93)

CASE 2: $\mathcal{P}(b) \neq \mathcal{P}(b') \wedge \mathcal{P}(b) \neq \mathcal{P}(n)$

$\lceil b \rceil < \lfloor b' \rfloor$	$b \rightarrow b'$, $\mathcal{P}(b) \neq \mathcal{P}(b')$ and definition 3.4 (3)	(3.94)
$\lceil \hat{b} \rceil = \lceil b \rceil$	$\mathcal{P}(b) \neq \mathcal{P}(n)$ and (3.85)	(3.95)
$\lfloor \hat{b} \rfloor > \lceil \hat{b} \rceil$	(3.94) and (3.95)	(3.96)
$\lfloor b' \rfloor \leq \beta$	definition 3.10	(3.97)
$\lfloor \hat{b'} \rfloor = \lfloor b' \rfloor \vee \lfloor \hat{b'} \rfloor = \beta + 1$	(3.86)	(3.98)
$\lfloor \hat{b'} \rfloor \geq \lfloor b' \rfloor$	(3.97) and (3.98)	(3.99)
$\lceil \hat{b} \rceil < \lfloor \hat{b'} \rfloor$	(3.96) and (3.99)	(3.100)
$\hat{b} \rightarrow \hat{b'}$	(3.100) and definition 3.4 (3)	(3.101)

CASE 3: $\mathcal{P}(b) \neq \mathcal{P}(b') \wedge \mathcal{P}(b) = \mathcal{P}(n)$

$\lceil b \rceil < \lfloor b' \rfloor$	$b \rightarrow b'$, $\mathcal{P}(b) \neq \mathcal{P}(b')$ and definition 3.4 (3)	(3.102)

$\lfloor b' \rfloor \leq \beta$	definition 3.10	(3.103)
$\lceil \hat{b} \rceil = \lceil b \rceil$	$b \rightarrow b'$, $\mathcal{P}(b) \neq \mathcal{P}(b')$, (3.103) and (3.85)	(3.104)
$\lceil \hat{b} \rceil \leq \lfloor b' \rfloor$	(3.102) and (3.104)	(3.105)
$\lfloor \hat{b'} \rfloor \geq \lfloor b' \rfloor$	$\mathcal{P}(b') \neq \mathcal{P}(n)$ and (3.86)	(3.106)
$\lceil \hat{b} \rceil < \lfloor \hat{b'} \rfloor$	(3.105) and (3.106)	(3.107)
$\hat{b} \rightarrow \hat{b'}$	(3.107) and definition 3.4 (3)	(3.108)

We have accounted for all possible cases, therefore, $\hat{b} \rightarrow \hat{b'}$. ■

Lemma 3.12 proves that the concurrent relationships in Ψ among blocks of partition B are accurately represented in $\widehat{\Psi}$. From the definition of concurrency we know that the events in question must have been executed by different processes. Therefore at most one of the events could have been executed by the same process as that which executed n. We arbitrarily choose b' and assume that it was not executed by the same process as that which executed n. Without assumptions concerning the process or original placement of b, we can conclude that the concurrency is properly shown in $\widehat{\Psi}$. We base our proof on the definition of a concurrency map as it relates to the display of concurrent events.

Lemma 3.12 $\forall b, b' \in B : b \| b' \Rightarrow \hat{b} \| \hat{b'}$.

Proof:

$\mathcal{P}(b) \neq \mathcal{P}(b')$	$b \| b'$ and definition 2.7	(3.109)
$\mathcal{P}(b') \neq \mathcal{P}(n)$	arbitrary assumption	(3.110)

$\lfloor \hat{b}' \rfloor = \lfloor b' \rfloor$	(3.110) and (3.86)	(3.111)
$\lceil \hat{b}' \rceil = \lceil b' \rceil$	(3.110) and (3.85)	(3.112)
$\lceil b \rceil \leq \beta$	definition 3.10	(3.113)
$\lceil \hat{b} \rceil = \lceil b \rceil \vee \lceil \hat{b} \rceil = \beta + 1$	(3.85)	(3.114)
$\lceil \hat{b} \rceil \geq \lceil b \rceil$	(3.113) and (3.114)	(3.115)
$\lceil b' \rceil \geq \lfloor b \rfloor$	$b' \| b$ and corollary 3.1	(3.116)
$\lfloor \hat{b} \rfloor = \lfloor b \rfloor$	(3.109), (3.116) and (3.86)	(3.117)
$\mathcal{I}(\hat{b}) \cap \mathcal{I}(\hat{b}') \neq \emptyset$	(3.111), (3.112), (3.115) and (3.117)	(3.118)
$\hat{b} \| \hat{b}'$	(3.118) and definition 3.4 (4)	(3.119)

■

In addition to the relationships of the blocks of B, we can show that all blocks of B are shown causally preceding n in $\widehat{\Psi}$. We are given that the placement of n is between rows $\beta + 1$ and $\alpha + 1$. The definition of α tells us that it is greater than β in all cases. Therefore the minimum of the image of n is $\beta + 1$. The definition of β is such that no block in B is found lower in Ψ and the transformation allows only those blocks executed by the same process as n to extend into $\beta + 1$. We have assumed that the temporal order of blocks in a process is not reversed, so the transformation must properly show the causal relationship. The proof of lemma 3.13 is composed of two cases. The first assumes that blocks b and n are executed by different processes. The second assumes that they were executed by the same process.

Lemma 3.13 $\forall b \in B : \hat{b} \rightarrow n$.

Proof:

CASE 1: $\mathcal{P}(b) \neq \mathcal{P}(n)$.

$\lceil b \rceil \leq \beta$	definition (3.10)	(3.120)
$\lceil \hat{b} \rceil = \lceil b \rceil$	$\mathcal{P}(b) \neq \mathcal{P}(n)$ and (3.85)	(3.121)
$\lceil \hat{b} \rceil \leq \beta$	(3.120) and (3.121)	(3.122)
$\lfloor n \rfloor = \beta + 1$	(3.84)	(3.123)
$\lceil \hat{b} \rceil < \lfloor n \rfloor$	(3.122) and (3.123)	(3.124)
$\hat{b} \rightarrow n$	(3.124) and definition 3.4 (3)	(3.125)

CASE 2: $\mathcal{P}(b) = \mathcal{P}(n)$.

$\lceil b \rceil \leq \beta$	definition 3.10	(3.126)
$\lceil \hat{b} \rceil = \lceil b \rceil \vee \lceil \hat{b} \rceil = \beta + 1$	(3.85)	(3.127)
$\lceil \hat{b} \rceil \leq \beta + 1$	(3.126) and (3.127)	(3.128)
$\lfloor n \rfloor = \beta + 1$	(3.84)	(3.129)
$\lceil \hat{b} \rceil \leq \lfloor n \rfloor$	(3.128) and (3.129)	(3.130)
$\hat{b} \rightarrow n$	$\mathcal{P}(b) = \mathcal{P}(n)$, (3.130) and definition 3.4 (2)	(3.131)

We have accounted for all possible cases, therefore, $\hat{b} \rightarrow n$. ■

We have shown that in all cases, the relationships among any two blocks of partition B are maintained in the transformation. In addition, we have shown that the causal re-

lationship between a block in B and n is properly displayed in the new concurrency map. The next section will consider the relationships among multiple blocks of partition A, the causal relationship between a block of partition B and a block of partition A, and the causal relationship between the n and a block from A.

3.4.2 Transforming the blocks of partition A

Transforming the images of blocks causally succeeding n is performed in a manner similar to those preceding n. Instead of retaining the original position of most blocks, we translate their images down by two rows to make room for the placement of n. Since the blocks were originally found at or below row α, they are moved to or below row $\alpha + 2$.

Note that the definitions of β and α do not require that α be exactly one larger than β. It could be that $\alpha \gg \beta$, in which case the translation would not be required. Room would already exist between the blocks of B and the blocks of A in the map for the placement of n. The transformation equations could be extended with more elaborate cases to accommodate this circumstance and produce a transformed graph of minimal size. We have opted for the less complicated transformation that still produces a correct concurrency map.

The minimum of the image of a block causally succeeding n is generally translated down by two rows. It is possible that a block $c \in C$ could be placed in $\widehat{\Psi}$ such that the maximum of its image is in row $\alpha + 1$. If c is concurrent to some $a \in A$ then the image of a must also include row $\alpha + 1$. In this case $\lfloor \hat{a} \rfloor$ is assigned the value $\alpha + 1$. However, some conditions prevent this assignment.

Consider the situation shown in figure 3.25(a). The image of a'' must continue to display the causal relationship with n after it has been translated to its new position. So

the minimum of the new image of a'' must be $\alpha + 2$. With this in mind, consider a'. To show the concurrency with a'', the translated image of a' must extend into $\alpha + 2$ as well. To avoid overlapping the translated images of a' and a, we are forced to assign $\alpha + 2$ to $\lfloor a \rfloor$.

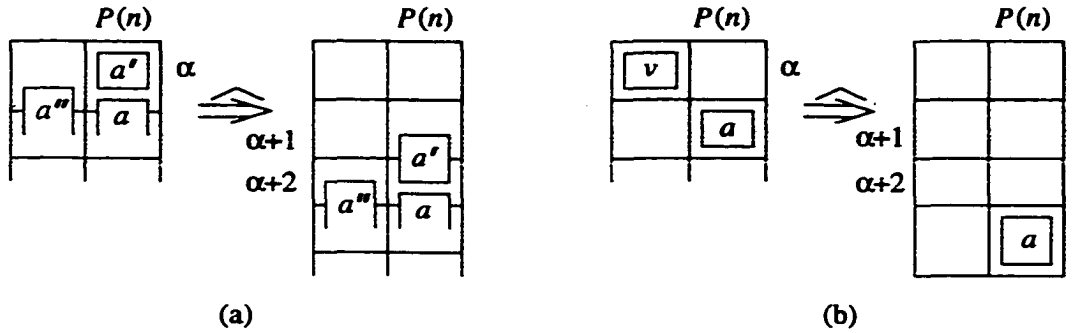


Figure 3.25: Special cases in the transformation of $a \in A$

Figure 3.25(b) demonstrates another condition under which the transformation must revert to the default assignment. A block v from either A or C causally precedes event a in a process other than $\mathcal{P}(a)$. Blocks that proceed a in $\mathcal{P}(a)$ are not considered. If v is found at or below row α in Ψ, then we must again revert to the default assignment.

If v is a block in A, then it will be placed such that the minimum of its image is below $\alpha + 1$. Since a and v are executed by different processes, the minimum of the transformed image of v must have moved down by two rows. To properly display the causal relationship between v and a, the image of a must be below the image of v which will not be the case if $\lfloor \hat{a} \rfloor$ is assigned the value $\alpha + 1$. Equation 3.132 formalizes the transformation.

$$\lfloor \hat{a} \rfloor = \begin{cases} \alpha + 1 & \text{if } [\mathcal{P}(a) = \mathcal{P}(n)] \\ & \wedge\, [\nexists a', a'' \in A : \mathcal{P}(a') = \mathcal{P}(a) \neq \mathcal{P}(a'') \wedge a' \rightarrow a \wedge \lfloor a'' \rfloor \leq \lceil a' \rceil] \\ & \wedge\, [\nexists v \in A \cup C : \mathcal{P}(v) \neq \mathcal{P}(a) \wedge v \rightarrow a \wedge \lceil v \rceil \geq \alpha] \\ \lfloor a \rfloor + 2 & \text{otherwise} \end{cases} \tag{3.132}$$

When constructing the maximum of the image of $\hat{a}$, we must consider the transformation of the minimum of the image. In most cases, those where a is executed by a process other than the process executing n, we translate downward by two rows. If a was executed by the same process as n, then we must consider other blocks as well.

If two blocks, a and a', are found in $\mathcal{P}(n)$, it may be necessary to have $\alpha+1$ in the images of both. Suppose that $a \rightarrow a'$ and both are concurrent to some $c \in C$. The transformed image of c will never exceed $\alpha+1$. To continue to show the concurrency between a' and c, we must set $\lfloor \hat{a}' \rfloor$ to $\alpha+1$. This implies that $\lceil \hat{a} \rceil$ is also $\alpha+1$.

Suppose instead that a and a' are executed by different processes and are concurrently related. Since, by definition, a' causally succeeds n, we must make $\lceil \hat{a}' \rceil$ at least $\alpha+2$. If we allow $\lfloor \hat{a} \rfloor$ to be $\alpha+1$ we will incorrectly show a causal relationship between a and a'. The inequality $\lfloor a'' \rfloor \leq \lceil a \rceil$ of equation 3.133 identifies those blocks that either causally precede or are concurrent to a.

$$\lceil \hat{a} \rceil = \begin{cases} \alpha+1 & \text{if } [\mathcal{P}(a) = \mathcal{P}(n)] \\ & \wedge\, [\exists a' \in A : \mathcal{P}(a') = \mathcal{P}(a) \wedge a \rightarrow a'] \\ & \wedge\, [\nexists a'' \in A : \mathcal{P}(a'') \neq \mathcal{P}(a) \wedge\ \lfloor a'' \rfloor \leq \lceil a \rceil] \\ \lceil a \rceil + 2 & \text{otherwise} \end{cases} \tag{3.133}$$

Any two blocks are either causally or concurrently related. Lemma 3.14 proves that a causal relationship between two blocks a and a' from partition A is accurately preserved in the transformation. The proof is divided into three cases. The first case assumes that a and a' are executed by the same process. This assumption allows us to be certain about their relative placement after transformation. Although the necessary information is not known to determine the exact placement of the causally succeeding block, we can determine a range

over which it may be placed. That range is sufficient to show the relationship is properly maintained. The second and third cases assume that a and a' are executed by different processes. Case two also assumes that a and n are also executed by different processes, while case three assumes the opposite.

Lemma 3.14 $\forall a, a' \in A : a \rightarrow a' \Rightarrow \hat{a} \rightarrow \hat{a}'$.

Proof:

CASE 1: $\mathcal{P}(a) = \mathcal{P}(a')$

$\lceil a \rceil \leq \lfloor a' \rfloor$	$\mathcal{P}(a) = \mathcal{P}(a')$ and definition 3.4 (2)	(3.134)
$\lceil a \rceil \geq \alpha$	definition 3.11	(3.135)
$\lceil \hat{a} \rceil = \alpha + 1 \vee \lceil \hat{a} \rceil = \lceil a \rceil + 2$	(3.133)	(3.136)
$\lceil \hat{a} \rceil \leq \lceil a \rceil + 2$	(3.135) and (3.136)	(3.137)
$\lfloor \hat{a}' \rfloor = \lfloor a' \rfloor + 2$	$a \rightarrow a'$ and (3.132)	(3.138)
$\lceil \hat{a} \rceil \leq \lfloor \hat{a}' \rfloor$	(3.134), (3.137) and (3.138)	(3.139)
$\hat{a} \rightarrow \hat{a}'$	(3.139) and definition 3.4 (2)	(3.140)

CASE 2: $\mathcal{P}(a) \neq \mathcal{P}(a') \wedge \mathcal{P}(a) \neq \mathcal{P}(n)$

$\lceil a \rceil < \lfloor a' \rfloor$	$a \rightarrow a'$, $\mathcal{P}(a) \neq \mathcal{P}(a')$ and definition 3.4 (3)	(3.141)
$\lceil \hat{a} \rceil = \lceil a \rceil + 2$	$\mathcal{P}(a) \neq \mathcal{P}(n)$ and (3.133)	(3.142)
$\lceil a \rceil \geq \alpha$	definition 3.11	(3.143)

$\lfloor \hat{a}' \rfloor = \lfloor a' \rfloor + 2$	$a \rightarrow a'$, $\mathcal{P}(a) \neq \mathcal{P}(a')$, (3.143) and (3.132)	(3.144)
$\lceil \hat{a} \rceil < \lfloor \hat{a}' \rfloor$	(3.141), (3.142) and (3.144)	(3.145)
$\hat{a} \rightarrow \hat{a}'$	(3.145) and definition 3.4 (3)	(3.146)

CASE 3: $\mathcal{P}(a) \neq \mathcal{P}(a') \wedge \mathcal{P}(a) = \mathcal{P}(n)$

$\lceil a \rceil < \lfloor a' \rfloor$	$a \rightarrow a'$, $\mathcal{P}(a) \neq \mathcal{P}(a')$ and definition 3.4 (3)	(3.147)
$\lceil a \rceil \geq \alpha$	definition 3.11	(3.148)
$\lceil \hat{a} \rceil = \alpha + 1 \vee \lceil \hat{a} \rceil = \lceil a \rceil + 2$	(3.132)	(3.149)
$\lceil \hat{a} \rceil \leq \lceil a \rceil + 2$	(3.148) and (3.149)	(3.150)
$\lfloor \hat{a}' \rfloor = \lfloor \hat{a}' \rfloor + 2$	$\mathcal{P}(a') \neq \mathcal{P}(n)$ and (3.133)	(3.151)
$\lceil \hat{a} \rceil < \lfloor \hat{a}' \rfloor$	(3.147), (3.150) and (3.151)	(3.152)
$\hat{a} \rightarrow \hat{a}'$	(3.152) and definition 3.4 (3)	(3.153)

We have accounted for all possible cases, therefore, $\hat{a} \rightarrow \hat{a}'$. ■

The accurate transformation of concurrent blocks of partition A is proven in lemma 3.15. Since the blocks are known to be concurrent, we are assured that at least one block is executed by a process other than the process which executed n. In the proof, we arbitrarily assume that a' is executed by a different process than n and show that the concurrency remains accurately displayed after the transformation. The definition of a concurrency map tells us that the intersection of the images of concurrent blocks is non-empty. Our proof is based on that fact and shows that transformation forces the intersection of the images to retain a common element.

Lemma 3.15 $\forall a, a' \in A : a \| a' \Rightarrow \hat{a} \| \hat{a}'$.

Proof:

$\mathcal{P}(a) \neq \mathcal{P}(a')$	$a \| a'$ and definition 2.7	(3.154)
$\mathcal{P}(a') \neq \mathcal{P}(n)$	arbitrary assumption	(3.155)
$\lfloor \hat{a}' \rfloor = \lfloor a' \rfloor + 2$	(3.155) and (3.132)	(3.156)
$\lceil \hat{a}' \rceil = \lceil a' \rceil + 2$	(3.155) and (3.133)	(3.157)
$\lfloor a \rfloor \geq \alpha$	definition 3.11	(3.158)
$\lfloor \hat{a} \rfloor = \alpha + 1 \vee \lfloor \hat{a} \rfloor = \lfloor a \rfloor + 2$	(3.132)	(3.159)
$\lfloor \hat{a} \rfloor \leq \lfloor a \rfloor + 2$	(3.158) and (3.159)	(3.160)
$\lfloor a' \rfloor \leq \lceil a \rceil$	$a \| a'$ and corollary 3.1	(3.161)
$\lceil \hat{a} \rceil = \lceil a \rceil + 2$	(3.154), (3.161) and (3.133)	(3.162)
$\mathcal{I}(\hat{a}) \cap \mathcal{I}(\hat{a}') \neq \emptyset$	(3.156), (3.157), (3.160) and (3.162)	(3.163)
$\hat{a} \| \hat{a}'$	(3.163) and definition 3.4 (4)	(3.164)

■

When n is added to the concurrency map, it is placed in a specific location. Namely, where the maximum of its image is in row $\alpha+1$. Only when a block of partition A is executed by the same process as n is it possibly allowed to extend into row $\alpha + 1$. By assuming that the temporal order of blocks is maintained through the transformation we insure that the causal relation is also maintained. Blocks of A executed by other processes, or that do not fulfill the other requirements for inclusion in row $\alpha + 1$, are moved to rows below $\alpha + 1$.

These blocks are shown to causally follow n after the transformation. Lemma 3.16 proves that all blocks of A are shown to causally follow n in $\widehat{\Psi}$.

Lemma 3.16 $\forall a \in A : n \rightarrow \hat{a}$.

Proof:

CASE 1: $\mathcal{P}(a) \neq \mathcal{P}(n)$.

$\lfloor a \rfloor \geq \alpha$	definition 3.11	(3.165)
$\lfloor \hat{a} \rfloor = \lfloor a \rfloor + 2$	$\mathcal{P}(a) \neq \mathcal{P}(n)$ and (3.132)	(3.166)
$\lfloor \hat{a} \rfloor \geq \alpha + 2$	(3.165) and (3.166)	(3.167)
$\lceil n \rceil = \alpha + 1$	(3.84)	(3.168)
$\lceil n \rceil < \lfloor \hat{a} \rfloor$	(3.167) and (3.168)	(3.169)
$n \rightarrow \hat{a}$	(3.169) and definition 3.4 (3)	(3.170)

CASE 2: $\mathcal{P}(a) = \mathcal{P}(n)$.

$\lfloor a \rfloor \geq \alpha$	definition 3.11	(3.171)
$\lfloor \hat{a} \rfloor = \alpha + 1 \vee \lfloor \hat{a} \rfloor = \lfloor a \rfloor + 2$	(3.132)	(3.172)
$\lfloor \hat{a} \rfloor \geq \alpha + 1$	(3.171) and (3.172)	(3.173)
$\lceil n \rceil = \alpha + 1$	(3.84)	(3.174)
$\lceil n \rceil \leq \lfloor \hat{a} \rfloor$	(3.173) and (3.174)	(3.175)
$n \rightarrow \hat{a}$	(3.175) and definition 3.4 (2)	(3.176)

We have accounted for all possible cases, therefore, $n \rightarrow \hat{a}$. ■

Causal relationships between the blocks of B and A are easily shown to be retained in the transformation as given in lemma 3.17. The definitions of β and α tell us that α is always larger than β. Furthermore, the blocks of B after transformation do not trespass into rows below $\beta + 1$ and the transformation of blocks in A forces them into rows no higher than $\alpha + 1$. These three facts provide all that is necessary to prove that the causal relationships found between B and A in Ψ are also found in $\widehat{\Psi}$.

Lemma 3.17 $\forall b \in B, a \in A : b \rightarrow a \Rightarrow \hat{b} \rightarrow \hat{a}$.

Proof:

$\lceil b \rceil \leq \beta$	definition 3.10	(3.177)
$\lceil \hat{b} \rceil = \beta + 1 \vee \lceil \hat{b} \rceil = \lceil b \rceil$	(3.85)	(3.178)
$\lceil \hat{b} \rceil \leq \beta + 1$	(3.177) and (3.178)	(3.179)
$\lfloor a \rfloor \geq \alpha$	definition 3.11	(3.180)
$\lfloor \hat{a} \rfloor = \alpha + 1 \vee \lfloor \hat{a} \rfloor = \lfloor a \rfloor + 2$	(3.132)	(3.181)
$\lfloor \hat{a} \rfloor \geq \alpha + 1$	(3.180) and (3.181)	(3.182)
$\beta < \alpha$	definition 3.11	(3.183)
$\lceil \hat{b} \rceil < \lfloor \hat{a} \rfloor$	(3.179), (3.182) and (3.183)	(3.184)
$\hat{b} \rightarrow \hat{a}$	(3.184) and definition 3.4 (3)	(3.185)

■

We have now partially shown that blocks of partition A are correctly transformed by equations (3.132) and (3.133). The inter-partition relationships, both causal and concurrent, have been proven correct as have the causal relationships between blocks in B and blocks in A. The placement of n has been shown to be correct with respect to A as well. The next section defines the transformation of blocks that are concurrent to n, the blocks of C, and uses those transformations as a basis for proving the remaining relationships are correctly transformed.

3.4.3 Transforming the blocks of partition C

Transformation of the images of blocks concurrent to n is accomplished by ensuring that the image of the concurrent blocks includes a row in which n will be placed. Equations (3.186) and (3.187) give the transformation of the minimum and maximum of the images of a block in partition C based on its original position and the positions of other blocks also in C.

In order for the blocks of partition C to be shown concurrent to n, their images must be at least partially contained in the range $\beta + 1$ to $\alpha + 1$. The transformation of C is such that some portion of the image is in the defined range.

Equation (3.186) gives the transformation of the minimum of the image of a block in C. Notice that the default action is to retain the original position. If the minimum falls between β and α, we simply translate the image down by one row. An original image outside the range must be stretched to show concurrency with n. Knowing that the maximum of the image of n is $\alpha + 1$ tells us that we must pull the minimum of the image of c back to $\alpha + 1$ if it is found after α.

Other factors must be considered if the minimum of the image of c is found before β as expressed in the first case of the equation. We can deduce that no event from B causally follows c, and we know that the original concurrency map is accurate. If no other event of C causally precedes c then there is no need to alter the minimum of c's image. Assume that some c' precedes c. From the definition of β, we know there exists some b concurrent to both c and c'. Specifically, there must exist a b causing the value of β to be non-zero. If b is executed by a process other than $\mathcal{P}(n)$, or if c and c' are executed by different processes, or both, then $\{b, c, c', n\}$ forms a forbidden q^*. Therefore we know that b and n are executed by one process and c and c' are executed by another process. Furthermore, as shown in lemma 3.7, there can exist no other blocks in B concurrent to c without forming a similar q^* subgraph. Figure 3.26 (a) shows a case where the minimum of c must be moved to $\beta + 1$ to allow the maximum of c' to also show concurrency with n. The relationships with b are also preserved through the transformation of B and C.

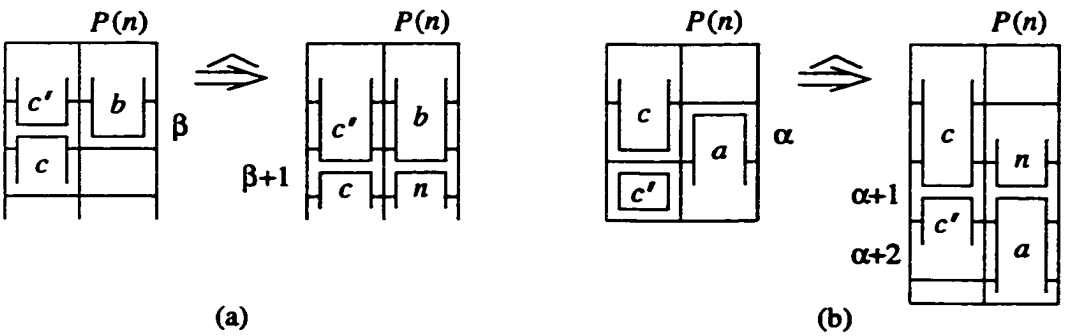


Figure 3.26: Special cases in the transformation of $c \in C$

$$\lfloor \hat{c} \rfloor = \begin{cases} \beta + 1 & \text{if } [\lfloor c \rfloor \leq \beta] \wedge [\exists c' \in C : c' \rightarrow c] \\ \lfloor c \rfloor + 1 & \text{if } [\beta < \lfloor c \rfloor < \alpha] \\ \alpha + 1 & \text{if } [\lfloor c \rfloor \geq \alpha] \\ \lfloor c \rfloor & \text{otherwise} \end{cases} \tag{3.186}$$

Transformation of the last row of the image of c is similarly accomplished. If $\lceil c \rceil$ is found in the concurrency map prior to β, then we know the concurrency between c and n cannot be shown unless the image of c is stretched to include $\beta + 1$. If $\lceil c \rceil$ is between β and α, then we simply increase $\lceil \hat{c} \rceil$ to $\lceil c \rceil + 1$ in order to retain the relationships with other blocks of partition C. If the maximum of the image of c is at least as great as α, we must again consider other blocks of C.

As shown in figure 3.26 (b), it is possible that another block c' causally follows c. See lemma 3.9 for a proof concerning the process placement of the constituent blocks. In order to show the concurrency between c' and n in the transformed concurrency map, we must allow the minimum of c' to be $\alpha + 1$. To avoid overlapping images, the implication is that the maximum of the image of c must be $\alpha + 1$ as well. If c' does not exist, the maximum of c is translated down by 2 rows to retain possible relationships with blocks from a.

$$\lceil \hat{c} \rceil = \begin{cases} \beta + 1 & \text{if } [\lceil c \rceil \leq \beta] \\ \lceil c \rceil + 1 & \text{if } [\beta < \lceil c \rceil < \alpha] \\ \alpha + 1 & \text{if } [\lceil c \rceil \geq \alpha] \wedge [\exists c' \in C : c \rightarrow c'] \\ \lceil c \rceil + 2 & \text{otherwise} \end{cases} \tag{3.187}$$

Lemma 3.18 shows that causal relationships between blocks of partition C remain unchanged through the transformation. The proof considers five cases based on the placement of blocks of Ψ.

Lemma 3.18 $\forall c, c' \in C : c \to c' \Rightarrow \hat{c} \to \hat{c'}$.

Proof:

CASE 1: $\lfloor c' \rfloor \leq \beta$

$\exists b \in B : \lceil b \rceil = \beta$	$\lfloor c' \rfloor \leq \beta$ and definition 3.10	(3.188)
$c' \not\to b$	property 3.3	(3.189)
$\lfloor c' \rfloor \leq \lceil b \rceil$	$\lfloor c' \rfloor \leq \beta$ and (3.188)	(3.190)
$b \not\to c'$	(3.190) and definition 3.4 (3)	(3.191)
$b \| c'$	(3.189), (3.191) and definition 2.7	(3.192)
$\mathcal{P}(c) = \mathcal{P}(c')$	$c \to c'$, (3.192) and lemma 3.8	(3.193)
$\lceil c \rceil \leq \lfloor c' \rfloor$	$c \to c'$, (3.193) and definition 3.4 (2)	(3.194)
$\lceil c \rceil \leq \beta$	$\lfloor c' \rfloor \leq \beta$ and (3.194)	(3.195)
$\lceil \hat{c} \rceil = \beta + 1$	(3.195) and (3.187)	(3.196)
$\lfloor \hat{c'} \rfloor = \beta + 1$	$\lfloor c' \rfloor \leq \beta$, $c \to c'$ and (3.186)	(3.197)
$\lceil \hat{c} \rceil = \lfloor \hat{c'} \rfloor$	(3.196) and (3.197)	(3.198)
$\hat{c} \to \hat{c'}$	(3.193), (3.198) and definition 3.4 (2)	(3.199)

CASE 2: $\beta < \lfloor c' \rfloor < \alpha \wedge \mathcal{P}(c) = \mathcal{P}(c')$

$\lceil c \rceil \leq \lfloor c' \rfloor$	$c \to c'$, $\mathcal{P}(c) = \mathcal{P}(c')$, and definition 3.4 (2)	(3.200)

$\lceil c \rceil < \alpha$	$\lfloor c' \rfloor < \alpha$ and (3.200)	(3.201)
$\lceil \hat{c} \rceil = \max(\lceil c \rceil, \beta) + 1$	(3.201) and (3.187)	(3.202)
$\lfloor \tilde{c}' \rfloor = \lfloor c' \rfloor + 1$	$\beta < \lfloor c' \rfloor < \alpha$ and (3.186)	(3.203)
$\lfloor \tilde{c}' \rfloor \geq \lceil c \rceil + 1$	(3.200) and (3.203)	(3.204)
$\lfloor \tilde{c}' \rfloor > \beta + 1$	$\beta < \lfloor c' \rfloor$ and (3.203)	(3.205)
$\lfloor \tilde{c}' \rfloor \geq \max(\lceil c \rceil, \beta) + 1$	(3.204) and (3.205)	(3.206)
$\lceil \hat{c} \rceil \leq \lfloor \tilde{c}' \rfloor$	(3.202) and (3.206)	(3.207)
$\hat{c} \rightarrow \tilde{c}'$	(3.207) and definition 3.4 (2)	(3.208)

CASE 3: $\beta < \lfloor c' \rfloor < \alpha \wedge \mathcal{P}(c) \neq \mathcal{P}(c')$

$\lceil c \rceil < \lfloor c' \rfloor$	$c \rightarrow c'$, $\mathcal{P}(c) \neq \mathcal{P}(c')$, and definition 3.4 (3)	(3.209)
$\lceil c \rceil < \alpha$	$\lfloor c' \rfloor < \alpha$ and (3.209)	(3.210)
$\lceil \hat{c} \rceil = \max(\lceil c \rceil, \beta) + 1$	(3.210) and (3.187)	(3.211)
$\lfloor \tilde{c}' \rfloor = \lfloor c' \rfloor + 1$	$\beta < \lfloor c' \rfloor < \alpha$ and (3.186)	(3.212)
$\lfloor \tilde{c}' \rfloor > \lceil c \rceil + 1$	(3.209) and (3.212)	(3.213)
$\lfloor \tilde{c}' \rfloor > \beta + 1$	$\beta < \lfloor c' \rfloor$ and (3.212)	(3.214)
$\lfloor \tilde{c}' \rfloor > \max(\lceil c \rceil, \beta) + 1$	(3.213) and (3.214)	(3.215)
$\lceil \hat{c} \rceil < \lfloor \tilde{c}' \rfloor$	(3.211) and (3.215)	(3.216)
$\hat{c} \rightarrow \tilde{c}'$	(3.216) and definition 3.4 (3)	(3.217)

CASE 4: $\lfloor c' \rfloor \geq \alpha \wedge \lceil c \rceil < \alpha$

$\lfloor \hat{c}' \rfloor = \alpha + 1$	$\lfloor c' \rfloor \geq \alpha$ and (3.186)	(3.218)
$\lceil \hat{c} \rceil < \alpha + 1$	$\lceil c \rceil < \alpha$ and (3.187)	(3.219)
$\lceil \hat{c} \rceil < \lfloor \hat{c}' \rfloor$	(3.218) and (3.219)	(3.220)
$\hat{c} \rightarrow \hat{c}'$	(3.220) and definition 3.4 (3)	(3.221)

CASE 5: $\lceil c \rceil \geq \alpha$

$\exists a \in A : \lfloor a \rfloor = \alpha$	$\lceil c \rceil \geq \alpha$ and definition 3.11	(3.222)
$a \not\rightarrow c$	property 3.2	(3.223)
$\lceil c \rceil \geq \lfloor a \rfloor$	$\lceil c \rceil \geq \alpha$ and (3.222)	(3.224)
$c \not\rightarrow a$	(3.224) and definition 3.4 (3)	(3.225)
$a \| c$	(3.223), (3.225) and definition 2.7	(3.226)
$\mathcal{P}(c) = \mathcal{P}(c')$	$c \rightarrow c'$, (3.226) and lemma 3.9	(3.227)
$\lceil \hat{c} \rceil = \alpha + 1$	$\lceil c \rceil \geq \alpha$, $c \rightarrow c'$ and (3.187)	(3.228)
$\lceil c \rceil \leq \lfloor c' \rfloor$	$c \rightarrow c'$, (3.227) and definition 3.4 (2)	(3.229)
$\lfloor c' \rfloor \geq \alpha$	$\lceil c \rceil \geq \alpha$ and (3.229)	(3.230)
$\lfloor \hat{c}' \rfloor = \alpha + 1$	(3.230) and (3.186)	(3.231)
$\lfloor \hat{c}' \rfloor = \lfloor \hat{c}' \rfloor$	(3.228) and (3.231)	(3.232)
$\hat{c} \rightarrow \hat{c}'$	(3.227), (3.232) and definition 3.4 (2)	(3.233)

We have accounted for all possible cases, therefore, $\hat{c} \rightarrow \hat{c}'$. ■

In lemma 3.19, we assume that two blocks from partition C are concurrently related and show that the transformation preserves that relationship. Since Ψ is assumed to be accurate, we know the intersection of their images is not empty. We consider the five possible cases for the intersection of their images. Initially, we assume that the maximum of the image of one block is less than β which implies the intersection is also less than β. We then consider the case where β is an element of the intersection. The next case assumes that the intersection is between β and α but contains neither. We then consider the case where α is an element of the intersection. Our final case assumes that the minimum of one block is greater than α, implying that all elements of the intersection are greater than α. Note that the cases are not mutually exclusive. It could be the case that both β and α are included in the intersection of images. We do, however, consider all possibilities.

Lemma 3.19 $\forall c, c' \in C : c \| c' \Rightarrow \hat{c} \| \hat{c}'$.

Proof:

CASE 1: $\lceil c \rceil < \beta$

$\lceil \hat{c} \rceil = \beta + 1$	$\lceil c \rceil < \beta$ and (3.187)	(3.234)
$\lfloor c' \rfloor \leq \lceil c \rceil$	$c \| c'$ and corollary 3.1	(3.235)
$\lfloor c' \rfloor < \beta$	$\lceil c \rceil < \beta$ and (3.235)	(3.236)
$\lfloor \hat{c}' \rfloor \leq \beta + 1$	(3.236) and (3.186)	(3.237)
$\lceil \hat{c}' \rceil \geq \beta + 1$	(3.187)	(3.238)

$\mathcal{I}(\hat{c}) \cap \mathcal{I}(\hat{c}') \neq \emptyset$	(3.234), (3.237) and (3.238)	(3.239)
$\hat{c} \| \hat{c}'$	(3.239) and definition 3.4 (4)	(3.240)

CASE 2: $\beta \in \mathcal{I}(c) \cap \mathcal{I}(c')$

$\lfloor c \rfloor \leq \beta$	$\beta \in \mathcal{I}(c)$	(3.241)
$\lfloor \hat{c} \rfloor = \beta + 1 \vee \lfloor \hat{c} \rfloor = \lfloor c \rfloor$	(3.241) and (3.186)	(3.242)
$\lfloor \hat{c} \rfloor \leq \beta + 1$	(3.241) and (3.242)	(3.243)
$\lfloor c' \rfloor \leq \beta$	$\beta \in \mathcal{I}(c')$	(3.244)
$\lfloor \hat{c'} \rfloor = \beta + 1 \vee \lfloor \hat{c'} \rfloor = \lfloor c' \rfloor$	(3.244) and (3.186)	(3.245)
$\lfloor \hat{c'} \rfloor \leq \beta + 1$	(3.244) and (3.245)	(3.246)
$\lceil \hat{c} \rceil \geq \beta + 1$	(3.187)	(3.247)
$\lceil \hat{c'} \rceil \geq \beta + 1$	(3.187)	(3.248)
$\mathcal{I}(\hat{c}) \cap \mathcal{I}(\hat{c}') \neq \emptyset$	(3.243), (3.246), (3.247) and (3.248)	(3.249)
$\hat{c} \| \hat{c}'$	(3.249) and definition 3.4 (4)	(3.250)

CASE 3: $\beta < \mathcal{I}(c) \cap \mathcal{I}(c') < \alpha$

$\psi \in \mathcal{I}(c) \cap \mathcal{I}(c')$	assumption	(3.251)
$\psi < \alpha$	$\mathcal{I}(c) \cap \mathcal{I}(c') < \alpha$ and (3.251)	(3.252)
$\psi > \beta$	$\beta < \mathcal{I}(c) \cap \mathcal{I}(c')$ and (3.251)	(3.253)
$\lfloor c \rfloor \leq \psi$	(3.251) and definition 3.1	(3.254)

$\lfloor c \rfloor < \alpha$	(3.252) and (3.254)	(3.255)
$\lfloor \hat{c} \rfloor = \max(\beta, \lfloor c \rfloor) + 1$	(3.255) and (3.186)	(3.256)
$\lfloor \hat{c} \rfloor \leq \psi + 1$	(3.253), (3.254) and (3.256)	(3.257)
$\lfloor c' \rfloor \leq \psi$	(3.251) and definition 3.1	(3.258)
$\lfloor c' \rfloor < \alpha$	(3.252) and (3.258)	(3.259)
$\lfloor \hat{c}' \rfloor = \max(\beta, \lfloor c' \rfloor) + 1$	(3.259) and (3.186)	(3.260)
$\lfloor \hat{c}' \rfloor \leq \psi + 1$	(3.253), (3.258) and (3.260)	(3.261)
$\lceil c \rceil \geq \psi$	(3.251) and definition 3.1	(3.262)
$\lceil c \rceil > \beta$	(3.253) and (3.262)	(3.263)
$\lceil \hat{c} \rceil \geq \min(\alpha, \lceil c \rceil) + 1$	(3.263) and (3.187)	(3.264)
$\lceil \hat{c} \rceil \geq \psi + 1$	(3.252), (3.262) and (3.264)	(3.265)
$\lceil c' \rceil \geq \psi$	(3.251) and definition 3.1	(3.266)
$\lceil c' \rceil > \beta$	(3.253) and (3.266)	(3.267)
$\lceil \hat{c}' \rceil \geq \min(\alpha, \lceil c' \rceil) + 1$	(3.267) and (3.187)	(3.268)
$\lceil \hat{c}' \rceil \geq \psi + 1$	(3.252), (3.266) and (3.268)	(3.269)
$\mathcal{I}(c) \cap \mathcal{I}(c') \neq \emptyset$	(3.257), (3.261), (3.265) and (3.269)	(3.270)
$\hat{c} \| \hat{c}'$	(3.270) and definition 3.4 (4)	(3.271)

CASE 4: $\alpha \in \mathcal{I}(c) \cap \mathcal{I}(c')$

$\lceil c \rceil \geq \alpha$	$\alpha \in \mathcal{I}(c)$	(3.272)

$\lceil \hat{c} \rceil = \alpha + 1 \vee \lceil \hat{c} \rceil = \lceil c \rceil + 2$	(3.272) and (3.187)	(3.273)
$\lceil \hat{c} \rceil \geq \alpha + 1$	(3.272) and (3.273)	(3.274)
$\lceil c' \rceil \geq \alpha$	$\alpha \in \mathcal{I}(c')$	(3.275)
$\lceil \hat{c}' \rceil = \alpha + 1 \vee \lceil \hat{c}' \rceil = \lceil c' \rceil + 2$	(3.275) and (3.187)	(3.276)
$\lceil \hat{c}' \rceil \geq \alpha + 1$	(3.275) and (3.276)	(3.277)
$\lfloor \hat{c} \rfloor \leq \alpha + 1$	(3.186)	(3.278)
$\lfloor \hat{c}' \rfloor \leq \alpha + 1$	(3.186)	(3.279)
$\mathcal{I}(\hat{c}) \cap \mathcal{I}(\hat{c}') \neq \emptyset$	(3.274), (3.277), (3.278) and (3.279)	(3.280)
$\hat{c} \| \hat{c}'$	(3.280) and definition 3.4 (4)	(3.281)

CASE 5: $\lfloor c \rfloor > \alpha$

$\lfloor \hat{c} \rfloor = \alpha + 1$	$\lfloor c \rfloor > \alpha$ and (3.186)	(3.282)
$\lfloor \hat{c}' \rfloor \leq \alpha + 1$	(3.186)	(3.283)
$\lceil c' \rceil \geq \lfloor c \rfloor$	$c \| c'$ and corollary 3.1	(3.284)
$\lceil c' \rceil > \alpha$	$\lfloor c \rfloor > \alpha$ and (3.284)	(3.285)
$\lceil \hat{c}' \rceil \geq \alpha + 1$	(3.285) and (3.187)	(3.286)
$\mathcal{I}(\hat{c}) \cap \mathcal{I}(\hat{c}') \neq \emptyset$	(3.282), (3.285) and (3.286)	(3.287)
$\hat{c} \| \hat{c}'$	(3.287) and definition 3.4 (4)	(3.288)

We have accounted for all possible cases, therefore, $\hat{c} \| \hat{c}'$. ■

By definition, we know that blocks from C must be concurrent to n when the transformation is complete. We need only prove that the intersection of the transformed image of each c and the image of n is not empty. The conditions defined in corollary 3.1 are that the maximum of each image is less than the minimum of the other image. The proof demonstrates that this is the case for all blocks in C.

Lemma 3.20 $\forall c \in C : \hat{c} \| n$.

Proof:

$\lfloor n \rfloor = \alpha + 1$	(3.84)	(3.289)
$\lceil n \rceil = \beta + 1$	(3.84)	(3.290)
$\lfloor \hat{c} \rfloor \leq \alpha + 1$	(3.186)	(3.291)
$\lceil \hat{c} \rceil \geq \beta + 1$	(3.187)	(3.292)
$\mathcal{I}(n) \cap \mathcal{I}(\hat{c}) \neq \emptyset$	(3.289), (3.290), (3.291) and (3.292)	(3.293)
$\hat{c} \| n$	(3.293) and definition 3.4 (4)	(3.294)

■

With the exception of n, a block of C can be either causally or concurrently related to any other block in the system. We consider each case in turn and prove that the original relationship is retained in the transformation. We begin by examining the possible relationships between c and a block from B, and then consider the relationships between c and a block from A.

If b from B is causally related to c in Ψ, then $\hat{b}$ will be causally related to $\hat{c}$ in $\widehat{\Psi}$. This is proven in lemma 3.21. We know from property 3.3 that b happens before c. We first

consider c and b from the same process, then the two possibilities for c from a different process than b. First, c may be placed such that it is above β. That is, the maximum of the image of c is less than or equal to β. The alternative configuration is one in which the image of c is at least partially following β, i.e., the maximum of the image of c is greater than β.

Lemma 3.21 $\forall b \in B, c \in C : b \rightarrow c \Rightarrow \hat{b} \rightarrow \hat{c}$.

Proof:

CASE 1: $\mathcal{P}(c) = \mathcal{P}(b)$

$\mathcal{P}(c) \neq \mathcal{P}(n)$	$c \| n$ and definition 2.7	(3.295)
$\mathcal{P}(b) \neq \mathcal{P}(n)$	$\mathcal{P}(c) = \mathcal{P}(b)$ and (3.295)	(3.296)
$\lceil b \rceil \leq \lfloor c \rfloor$	$b \rightarrow c$ and definition 3.4 (2)	(3.297)
$\lceil \hat{b} \rceil = \lceil b \rceil$	(3.296) and (3.85)	(3.298)
$\lceil \hat{b} \rceil \leq \lfloor c \rfloor$	(3.297) and (3.298)	(3.299)
$\lceil b \rceil \leq \beta$	definition 3.10	(3.300)
$\lceil \hat{b} \rceil \leq \beta$	(3.298) and (3.300)	(3.301)
$\lfloor \hat{c} \rfloor = \lfloor c \rfloor \vee \lfloor \hat{c} \rfloor \geq \beta + 1$	(3.186)	(3.302)
$\lfloor \hat{c} \rfloor \geq \lceil \hat{b} \rceil$	(3.299), (3.301) and (3.302)	(3.303)
$\hat{b} \rightarrow \hat{c}$	(3.303) and definition 3.4 (2)	(3.304)

CASE 2: $\mathcal{P}(c) \neq \mathcal{P}(b) \wedge \lfloor c \rfloor \leq \beta$

$\lceil b \rceil < \lfloor c \rfloor$	$b \rightarrow c$, $\mathcal{P}(c) \neq \mathcal{P}(b)$ and definition 3.4 (3)	(3.305)
$\lceil \hat{b} \rceil = \lceil b \rceil$	$b \rightarrow c$, $\mathcal{P}(c) \neq \mathcal{P}(b)$, $\lfloor c \rfloor \leq \beta$ and (3.85)	(3.306)
$\lfloor \hat{c} \rfloor = \beta + 1 \vee \lfloor \hat{c} \rfloor = \lfloor c \rfloor$	$\lfloor c \rfloor \leq \beta$ and (3.186)	(3.307)
$\lfloor \hat{c} \rfloor \geq \lfloor c \rfloor$	$\lfloor c \rfloor \leq \beta$ and (3.307)	(3.308)
$\lceil \hat{b} \rceil < \lfloor \hat{c} \rfloor$	(3.305), (3.306) and (3.308)	(3.309)
$\hat{b} \rightarrow \hat{c}$	(3.309) and definition 3.4 (3)	(3.310)

CASE 3: $\mathcal{P}(c) \neq \mathcal{P}(b) \wedge \lfloor c \rfloor > \beta$

$\lceil b \rceil \leq \beta$	definition 3.10	(3.311)
$\lceil \hat{b} \rceil = \lceil b \rceil \vee \lceil \hat{b} \rceil = \beta + 1$	(3.85)	(3.312)
$\lceil \hat{b} \rceil \leq \beta + 1$	(3.311) and (3.312)	(3.313)
$\lfloor \hat{c} \rfloor = \lfloor c \rfloor + 1 \vee \lfloor \hat{c} \rfloor = \alpha + 1$	$\lfloor c \rfloor > \beta$ and (3.186)	(3.314)
$\alpha > \beta$	definition 3.11	(3.315)
$\lfloor \hat{c} \rfloor > \beta + 1$	$\lfloor c \rfloor > \beta$, (3.314) and (3.315)	(3.316)
$\lceil \hat{b} \rceil < \lfloor \hat{c} \rfloor$	(3.313) and (3.316)	(3.317)
$\hat{b} \rightarrow \hat{c}$	(3.317) and definition 3.4 (3)	(3.318)

We have accounted for all possible cases, therefore, $\hat{b} \rightarrow \hat{c}$. ■

In lemma 3.22 we show that the relationship between c and b is also maintained if it is concurrent rather than causal. The proof considers the two cases based on the existence

or absence of c' of C which causally precedes c. If c' does exist, then we can accurately determine the position of both $\hat{b}$ and $\hat{c}$ and show that concurrency is indeed displayed after the transformation. If, on the other hand, c' does not exist we are faced with multiple possibilities. However, all but one are discounted because they indicate the creation of a two-, three- or four-process q^* in $\widehat{\Psi}$.

Lemma 3.22 $\forall b \in B, c \in C : b \| c \Rightarrow \hat{b} \| \hat{c}$.

Proof:

$\lceil b \rceil \leq \beta$	definition 3.10	(3.319)
$\lfloor c \rfloor \leq \lceil b \rceil$	$b \| c$ and corollary 3.1	(3.320)
$\lfloor c \rfloor \leq \beta$	(3.319) and (3.320)	(3.321)

CASE 1: $\nexists c' \in C : c' \rightarrow c$

$\lfloor \hat{c} \rfloor = \lfloor c \rfloor$	$\nexists c' \in C : c' \rightarrow c$, (3.321) and (3.186)	(3.322)
$\lceil \hat{c} \rceil \geq \beta + 1$	(3.187)	(3.323)
$\lceil \hat{b} \rceil = \lceil b \rceil \vee \lceil \hat{b} \rceil = \beta + 1$	(3.85)	(3.324)
$\lceil \hat{b} \rceil \geq \lceil b \rceil$	(3.319) and (3.324)	(3.325)
$\lfloor \hat{b} \rfloor = \lfloor b \rfloor \vee \lfloor \hat{b} \rfloor = \beta + 1$	(3.86)	(3.326)
$\lfloor \hat{b} \rfloor \leq \beta + 1$	(3.319) and (3.326)	(3.327)
$\lfloor \hat{c} \rfloor \leq \lceil \hat{b} \rceil$	(3.320), (3.322) and (3.325)	(3.328)
$\lceil \hat{c} \rceil \geq \lfloor \hat{b} \rfloor$	(3.323) and (3.327)	(3.329)

$\hat{b}\|\|\hat{c}$	(3.328), (3.329) and corollary 3.1	(3.330)

CASE 2: $\exists c' \in C : c' \rightarrow c$

$\lfloor \hat{c} \rfloor \leq \beta$	$\exists c' \in C : c' \rightarrow c$, (3.331) and (3.186)	(3.331)
$\mathcal{P}(b) = \mathcal{P}(n)$	$c' \rightarrow c$, $b\|\|c$ and lemma 3.7	(3.332)
$c\|\|n$	definition 3.9	(3.333)
$\mathcal{P}(c) \neq \mathcal{P}(n)$	(3.333) and definition 2.7	(3.334)
$\mathcal{P}(c) = \mathcal{P}(c')$	$c' \rightarrow c$, $b\|\|c$ and lemma 3.7	(3.335)

We now show that the second clause of (3.85) must be true. Assume $\exists b' \in B : \mathcal{P}(b') = \mathcal{P}(b) \wedge b \rightarrow b'$. If this assumption is not valid, the clause would immediately be verified.

$\mathcal{P}(b') = \mathcal{P}(n)$	$\mathcal{P}(b') = \mathcal{P}(b)$ and (3.332)	(3.336)
$c' \not\rightarrow b'$	property 3.3	(3.337)
$\lceil c' \rceil \geq \lfloor b' \rfloor$	(3.337) and definition 3.4 (3)	(3.338)
$\lfloor c \rfloor \geq \lceil c' \rceil$	$c' \rightarrow c$, (3.335) and definition 3.4 (2)	(3.339)
$\lfloor c \rfloor \geq \lfloor b' \rfloor$	(3.338) and (3.339)	(3.340)
$\nexists b'' : \lceil b'' \rceil \geq \lfloor c \rfloor \wedge \mathcal{P}(b'') \neq \mathcal{P}(n)$	$c' \rightarrow c$ and lemma 3.8	(3.341)
$\nexists b'' : \lceil b'' \rceil \geq \lfloor b' \rfloor \wedge \mathcal{P}(b'') \neq \mathcal{P}(n)$	(3.340) and (3.341)	(3.342)
$\nexists b'' : \lceil b'' \rceil \geq \lfloor b' \rfloor \wedge \mathcal{P}(b'') \neq \mathcal{P}(b')$	(3.336) and (3.342)	(3.343)

To verify the third clause of (3.85), we refute the assumption $\exists v \in B \cup C : b \rightarrow v \wedge \lfloor v \rfloor \leq \beta \wedge \mathcal{P}(b) \neq \mathcal{P}(v)$

$\lceil b \rceil < \lfloor v \rfloor$	$b \rightarrow v$, $\mathcal{P}(b) \neq \mathcal{P}(v)$ and definition 3.4 (3)	(3.344)
$\lfloor c \rfloor \leq \lceil b \rceil$	$b \| v$ and definition 3.4 (4)	(3.345)
$\lfloor c \rfloor < \lfloor v \rfloor$	(3.344) and (3.345)	(3.346)
$\lfloor c \rfloor < \lceil v \rceil$	$\lfloor v \rfloor \leq \lceil v \rceil$ and (3.346)	(3.347)
$\mathcal{P}(v) \neq \mathcal{P}(n)$	$\mathcal{P}(b) \neq \mathcal{P}(v)$ and (3.332)	(3.348)
$v \notin B$	(3.347), (3.348) and lemma 3.8	(3.349)
$v \in C$	$v \in B \cup C$ and (3.349)	(3.350)
$\lceil b \rceil < \beta$	$\lfloor v \rfloor \leq \beta$ and (3.344)	(3.351)
$\exists b' : \lceil b' \rceil = \beta$	(3.351) and definition 3.10	(3.352)
$\lceil c' \rceil \leq \lfloor c \rfloor$	$c' \rightarrow c$, (3.335) and definition 3.4 (2)	(3.353)
$\lceil c' \rceil < \lfloor v \rfloor$	(3.346) and (3.353)	(3.354)
$c' \rightarrow v$	(3.354) and definition 3.4 (3)	(3.355)
$v \not\rightarrow b'$	(3.350) and property 3.3	(3.356)
$\lceil v \rceil \geq \lfloor b' \rfloor$	(3.356) and definition 3.4 (2)	(3.357)
$\lfloor v \rfloor \leq \lceil b' \rceil$	$\lfloor v \rfloor \leq \beta$ and (3.352)	(3.358)
$v \| b'$	(3.357), (3.358) and corollary 3.1	(3.359)
$\mathcal{P}(v) = \mathcal{P}(c')$	(3.350), (3.355), (3.359) and lemma 3.7	(3.360)
$c' \| n$	definition 3.9	(3.361)

$c \| n$	definition 3.9	(3.362)
$c \not\rightarrow b'$	property 3.3	(3.363)
$\lfloor c' \rfloor < \lceil b' \rceil$	(3.354) and (3.358)	(3.364)
$b' \not\rightarrow c'$	(3.364) and definition 3.4 (3)	(3.365)
$c' \| b'$	(3.363), (3.365) and definition 2.7	(3.366)
$\lfloor c \rfloor < \lceil b' \rceil$	(3.346) and (3.358)	(3.367)
$b' \not\rightarrow c$	(3.367) and definition 3.4 (3)	(3.368)
$c \| b'$	(3.363), (3.368) and definition 2.7	(3.369)
$\{c, c', v, b, b', n\} = q^*$	$b \rightarrow v$, (3.361), (3.362), (3.366) and (3.369)	(3.370)

Since we have assumed that no q^* exists, the supposition must be false. We have therefore verified all three clauses of (3.85).

$\lceil \hat{b} \rceil = \beta + 1$	(3.332), (3.343), (3.370) and (3.85)	(3.371)
$\lceil \hat{c} \rceil \geq \beta + 1$	(3.187)	(3.372)
$\mathcal{I}(\hat{b}) \cap \mathcal{I}(\hat{c}) \neq \emptyset$	(3.331), (3.371) and (3.372)	(3.373)
$\hat{b} \| \hat{c}$	(3.373) and definition 3.4 (4)	(3.374)

We have accounted for all possible cases, therefore, $\hat{b} \| \hat{c}$. ■

The following two lemmas show that the relationships between blocks of A and C are properly maintained in the transformation. We begin, in lemma 3.23, by assuming that the relationship is causal. From property 3.2 we know that there is only one possible causal relationship between c and a. Namely, c must happen before a. We consider the possible

placement of the maximum of the image of c in relation to the defined value of α. We first assume that c and a are executed by the same process. We then consider two possible arrangements when c and a are executed by different processes. Either the maximum of the image of c is greater than or equal to α, or it is less than α.

Lemma 3.23 $\forall c \in C, a \in A : c \rightarrow a \Rightarrow \hat{c} \rightarrow \hat{a}$.

Proof:

CASE 1: $\mathcal{P}(c) = \mathcal{P}(a)$

$\mathcal{P}(c) \neq \mathcal{P}(n)$	definition 3.9	(3.375)
$\mathcal{P}(a) \neq \mathcal{P}(n)$	$\mathcal{P}(c) = \mathcal{P}(a)$ and (3.375)	(3.376)
$\lceil c \rceil \leq \lfloor a \rfloor$	$c \rightarrow a$, $\mathcal{P}(c) = \mathcal{P}(a)$ and definition 3.4 (2)	(3.377)
$\lfloor \hat{a} \rfloor = \lfloor a \rfloor + 2$	(3.376) and (3.132)	(3.378)
$\lceil \hat{c} \rceil \leq \lceil c \rceil + 2 \vee \lceil \hat{c} \rceil = \beta + 1$	(3.187)	(3.379)
$\lceil \hat{c} \rceil \leq \lfloor a \rfloor + 2 \vee \lceil \hat{c} \rceil = \beta + 1$	(3.377) and (3.379)	(3.380)
$\lfloor a \rfloor \geq \alpha$	definition 3.11	(3.381)
$\lfloor a \rfloor \geq \beta + 1$	$\alpha > \beta$ and (3.381)	(3.382)
$\lceil \hat{c} \rceil \leq \lfloor a \rfloor + 2 \vee \lceil \hat{c} \rceil \leq \lfloor a \rfloor$	(3.380) and (3.382)	(3.383)
$\lceil \hat{c} \rceil \leq \lfloor \hat{a} \rfloor$	(3.383)	(3.384)
$\hat{c} \rightarrow \hat{a}$	(3.384) and definition 3.4 (2)	(3.385)

CASE 2: $\mathcal{P}(c) \neq \mathcal{P}(a) \wedge \lceil c \rceil < \alpha$

$\lfloor a \rfloor \geq \alpha$	definition 3.11	(3.386)
$\lfloor \hat{a} \rfloor = \alpha + 1 \vee \lfloor \hat{a} \rfloor = \lfloor a \rfloor + 2$	(3.132)	(3.387)
$\lfloor \hat{a} \rfloor \geq \alpha + 1$	(3.386) and (3.387)	(3.388)
$\lceil \hat{c} \rceil < \alpha + 1$	$\lceil c \rceil < \alpha$ and (3.187)	(3.389)
$\lceil \hat{c} \rceil < \lfloor \hat{a} \rfloor$	(3.388) and (3.389)	(3.390)
$\hat{c} \rightarrow \hat{a}$	(3.390) and definition 3.4 (3)	(3.391)

CASE 3: $\mathcal{P}(c) \neq \mathcal{P}(a) \wedge \lceil c \rceil \geq \alpha$

$\lceil c \rceil < \lfloor a \rfloor$	$\lceil c \rceil \geq \alpha$, $\mathcal{P}(c) \neq \mathcal{P}(a)$ and definition 3.4 (2)	(3.392)
$\lfloor \hat{a} \rfloor = \lfloor a \rfloor + 2$	$c \rightarrow a$, $\lceil c \rceil \geq \alpha$ and (3.132)	(3.393)
$\lceil \hat{c} \rceil \leq \lceil c \rceil + 2$	$\lceil c \rceil \geq \alpha$ and (3.187)	(3.394)
$\lceil \hat{c} \rceil < \lfloor \hat{a} \rfloor$	(3.392), (3.393) and (3.394)	(3.395)
$\hat{c} \rightarrow \hat{a}$	(3.395) and definition 3.4 (3)	(3.396)

We have accounted for all possible cases, therefore, $\hat{c} \rightarrow \hat{a}$. ■

Lemma 3.24 shows that a concurrent relationship between c and a is also maintained in the transformation. In a manner similar to that used to show the concurrency between c and b, we consider two cases. Either a block c' from C exists that causally precedes c, or it does not. In the first case, we can accurately determine the placement of the blocks in $\widehat{\Psi}$ and deduce the correct display of concurrency. In the second case, we can directly

determine the placement of c. To avoid creating a two-, three- or four-process q^* in $\widehat{\Psi}$, we can reduce the possibilities of the placement of $\hat{a}$.

Lemma 3.24 $\forall c \in C, a \in A : c \| a \Rightarrow \hat{c} \| \hat{a}$.

Proof:

$\lfloor a \rfloor \geq \alpha$	definition 3.11	(3.397)
$\lceil c \rceil \geq \lfloor a \rfloor$	$c \| a$ and corollary 3.1	(3.398)
$\lceil c \rceil \geq \alpha$	(3.397) and (3.398)	(3.399)

CASE 1: $\nexists c' \in C : c \rightarrow c'$

$\lfloor \hat{c} \rfloor \leq \alpha + 1$	(3.186)	(3.400)
$\lceil \hat{a} \rceil = \alpha + 1 \vee \lceil \hat{a} \rceil = \lceil a \rceil + 2$	(3.133)	(3.401)
$\lceil a \rceil \geq \alpha$	$\lceil a \rceil \geq \lfloor a \rfloor$ and (3.397)	(3.402)
$\lceil \hat{a} \rceil \geq \alpha + 1$	(3.401) and (3.402)	(3.403)
$\lceil \hat{a} \rceil \geq \lfloor \hat{c} \rfloor$	(3.400) and (3.403)	(3.404)
$\lceil \hat{c} \rceil = \lceil c \rceil + 2$	(3.399) and (3.187)	(3.405)
$\lceil \hat{c} \rceil \geq \lceil a \rceil + 2$	(3.398) and (3.405)	(3.406)
$\lfloor \hat{a} \rfloor = \alpha + 1 \vee \lfloor \hat{a} \rfloor = \lfloor a \rfloor + 2$	(3.132)	(3.407)
$\lfloor \hat{a} \rfloor \leq \lfloor a \rfloor + 2$	(3.397) and (3.407)	(3.408)
$\lfloor \hat{a} \rfloor \leq \lceil \hat{c} \rceil$	(3.406) and (3.408)	(3.409)

$\hat{c}\|\hat{a}$	(3.404) and (3.409)	(3.410)

CASE 2: $\exists c' \in C : c \rightarrow c'$

$\lceil \hat{c} \rceil = \alpha + 1$	(3.399), $\exists c' \in C : c \rightarrow c'$ and (3.187)	(3.411)
$\mathcal{P}(a) = \mathcal{P}(n)$	$c \rightarrow c'$, $c\|a$ and lemma 3.9	(3.412)
$c\|n$	definition 3.9	(3.413)
$\mathcal{P}(c) \neq \mathcal{P}(n)$	(3.413) and definition 2.7	(3.414)
$\mathcal{P}(c) = \mathcal{P}(c')$	$c \rightarrow c'$, $c\|a$ and lemma 3.9	(3.415)

We now show that the second clause of (3.132) must be true. Assume $\exists a' \in A : \mathcal{P}(a') = \mathcal{P}(a) \wedge a' \rightarrow a$. If this assumption is not valid, the clause would immediately be verified.

$\mathcal{P}(a') = \mathcal{P}(n)$	$\mathcal{P}(a') = \mathcal{P}(b)$ and (3.412)	(3.416)
$a' \not\rightarrow c'$	property 3.2	(3.417)
$\lceil a' \rceil \geq \lfloor c' \rfloor$	(3.417) and definition 3.4 (3)	(3.418)
$\lceil c \rceil \leq \lfloor c' \rfloor$	$c \rightarrow c'$, (3.415) and definition 3.4 (2)	(3.419)
$\lceil c \rceil \leq \lceil a' \rceil$	(3.418) and (3.419)	(3.420)
$\nexists a'' : \lfloor a'' \rfloor \leq \lceil c \rceil \wedge \mathcal{P}(a'') \neq \mathcal{P}(n)$	$c \rightarrow c'$ and lemma 3.10	(3.421)
$\nexists a'' : \lfloor a'' \rfloor \leq \lceil a' \rceil \wedge \mathcal{P}(a'') \neq \mathcal{P}(n)$	(3.420) and (3.421)	(3.422)
$\nexists a'' : \lfloor a'' \rfloor \leq \lceil a' \rceil \wedge \mathcal{P}(a'') \neq \mathcal{P}(a')$	(3.416) and (3.422)	(3.423)

To verify the third clause of (3.132), we assume it is true and then refute the assumption.

Assume $\exists v \in A \cup C : \mathcal{P}(v) \neq \mathcal{P}(a) \wedge v \rightarrow a \wedge \lceil v \rceil \geq \alpha$.

$\lceil v \rceil < \lfloor a \rfloor$	$v \rightarrow a$, $\mathcal{P}(v) \neq \mathcal{P}(a)$ and definition 3.4 (3)	(3.424)
$\lfloor a \rfloor \leq \lceil c \rceil$	$a \| v$ and definition 3.4 (4)	(3.425)
$\lfloor v \rfloor < \lfloor c \rfloor$	(3.424) and (3.425)	(3.426)
$\lfloor v \rfloor < \lceil c \rceil$	$\lfloor v \rfloor \leq \lceil v \rceil$ and (3.426)	(3.427)
$\mathcal{P}(v) \neq \mathcal{P}(n)$	$\mathcal{P}(v) \neq \mathcal{P}(a)$ and (3.412)	(3.428)
$v \notin A$	(3.427), (3.428) and lemma 3.10	(3.429)
$v \in C$	$v \in A \cup C$ and (3.429)	(3.430)
$\lfloor a \rfloor > \alpha$	$\lceil v \rceil \geq \alpha$ and (3.424)	(3.431)
$\exists a' : \lfloor a' \rfloor = \alpha$	(3.431) and definition 3.11	(3.432)
$\lceil c \rceil \leq \lfloor c' \rfloor$	$c \rightarrow c'$, (3.415) and definition 3.4 (2)	(3.433)
$\lceil v \rceil < \lfloor c' \rfloor$	(3.426) and (3.433)	(3.434)
$v \rightarrow c'$	(3.434) and definition 3.4 (3)	(3.435)
$a' \not\rightarrow v$	(3.430) and property 3.2	(3.436)
$\lceil a' \rceil \geq \lfloor v \rfloor$	(3.436) and definition 3.4 (2)	(3.437)
$\lceil v \rceil \geq \lfloor a' \rfloor$	$\lceil v \rceil \geq \alpha$ and (3.432)	(3.438)
$v \| a'$	(3.437), (3.438) and corollary 3.1	(3.439)
$\mathcal{P}(v) = \mathcal{P}(c')$	(3.430), (3.435), (3.438) and lemma 3.9	(3.440)
$c' \| n$	definition 3.9	(3.441)

$c \| n$	definition 3.9	(3.442)
$a' \not\rightarrow c$	property 3.2	(3.443)
$\lfloor a' \rfloor < \lceil c' \rceil$	(3.434) and (3.438)	(3.444)
$c' \not\rightarrow a'$	(3.444) and definition 3.4 (3)	(3.445)
$c' \| a'$	(3.443) and (3.445)	(3.446)
$\lfloor a' \rfloor < \lceil c \rceil$	(3.426) and (3.438)	(3.447)
$c \not\rightarrow a'$	(3.447) and definition 3.4 (3)	(3.448)
$c \| a'$	(3.443) and (3.448)	(3.449)
$\{c, c', v, a, a', n\} = q^*$	$v \rightarrow a$, (3.441), (3.442), (3.446) and (3.449)	(3.450)

Since we have assumed that no q^* exists, the assumption must be false. We have therefore verified all three clauses of (3.132).

$\lfloor \hat{a} \rfloor = \alpha + 1$	(3.412), (3.423), (3.450) and (3.132)	(3.451)
$\mathcal{I}(\hat{a}) \cap \mathcal{I}(\hat{c}) \neq \emptyset$	(3.411) and (3.451)	(3.452)
$\hat{a} \| \hat{c}$	(3.452) and definition 3.4 (4)	(3.453)

We have accounted for all possible cases, therefore, $\hat{b} \| \hat{c}$. ∎

In this section we have detailed the transformations required to insert a new block n into an accurate concurrency map to produce another accurate concurrency map. Both the original map and the new map accurately display the relationships between all included blocks if no q^*, either two-, three- or four-process, exists in the underlying concurrency graph. The next section expands on this to show that the connection between the q^* graph

and the concurrency map is that the existence of one is sufficient to imply the nonexistence of the other.

3.5 Significance of q*

The preceding barrage of lemmas has set the stage for the theorem 3.4. We have shown that all four-node subgraphs of a concurrency graph can be accurately displayed as a concurrency map. From here we construct an inductive proof based on the transformation equations of the previous section and their proofs of correctness. We show that if no q^* exists in a concurrency graph then it can be accurately displayed as a concurrency map.

Theorem 3.4 *If there does not exist a q^* induced subgraph of $\bar{\mathbf{H}}$ then an accurate concurrency map representation of $\mathbf{H}$ can be constructed.*

Proof: We proceed by induction.

Basis: All induced four-node subgraphs $\bar{h} \subseteq \bar{\mathbf{H}} : \bar{h} \neq q^*$ can be accurately represented as a concurrency map. This is proven by lemma 3.6

Induction: Assume that Ψ is an accurate concurrency map representation of a k-node subgraph of $\bar{\mathbf{H}}$ for some $k \geq 4$. We add node $n \in \bar{\mathbf{H}}$ to Ψ as directed by transformation equations (3.84), (3.85), (3.86), (3.132), (3.133), (3.186) and (3.187). The previous sections have proven that $\widehat{\Psi}$, when constructed in this manner, is an accurate concurrency map representation of a $k+1$-node subgraph of $\bar{\mathbf{H}}$ if the induced subgraph does not contain a q^*.

Therefore, an accurate concurrency map representation is possible for all executions where there does not exist a q^* induced subgraph of $\bar{\mathbf{H}}$. ■

We saw in theorem 3.3 that the construction of an accurate concurrency map representation of a distributed system execution is dependent on the absence of a q^* in the underlying concurrency graph. Together with theorem 3.4 which states that an accurate concurrency map representation is possible if no q^* exists in the concurrency graph, we can show the relationship between q^* and Ψ.

Theorem 3.5 *An accurate concurrency map depiction of a distributed system execution is possible if and only if there does not exist a q^* subgraph of the concurrent graph of the execution.*

Proof: Follows directly from theorems 3.3 and 3.4. ■

To determine whether or not an execution can be accurately displayed as a concurrency map, we need only determine whether or not a q^* subgraph exists in the concurrency graph. An algorithm to accomplish this feat will be exponential since it must construct and criticize each set of four and six events. However, that is a marked improvement over the challenge of creating and checking every possible concurrency map representation of the execution.

Although neither technique will likely be implemented, we have shown that a better way exists. If we assume that an execution exists where q^* cannot be found in the concurrency graph, then the technique can be used to construct an accurate representation of the distributed systems execution. Figure 3.27 demonstrates the transformation of Ψ into $\widehat{\Psi}$ with the inclusion of n. Notice that all blocks in B with the exception of those occurring in the same process as n retain their original position. Also notice that all blocks of A with the exception of those occurring in the same process as n are translated down by two rows but otherwise retain their relative order.

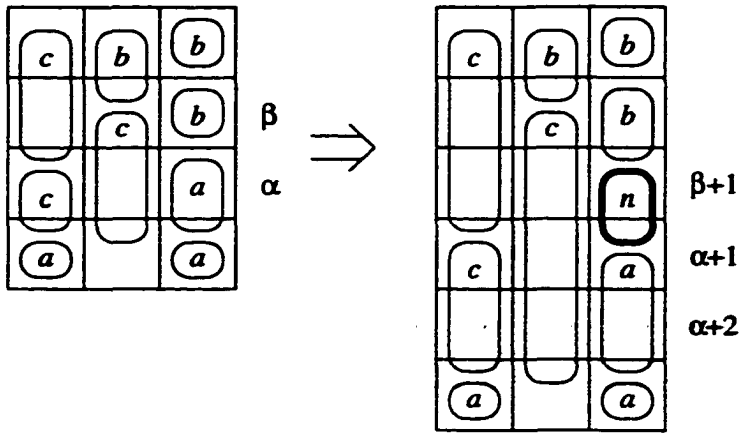


Figure 3.27: Transformation example

3.6 Evaluation

Stone's concurrency map displays the causality and concurrency of a distributed system in a clear and concise manner. Events that are causally related are shown separated by a vertical bar across the display. The technique also shows the possibility of concurrent execution of events by overlapping events in a single row.

However, under some conditions the concurrency map is not accurate. Through the proofs in the preceding sections, we have shown that the problems concerning the concurrency map are intrinsic. A two dimensional grid structure is inadequate for the display of the concurrency relation. The non-transitive nature of concurrency prevents representation using a geometric model.

In the next chapter we develop a technique that is theoretically accurate for the display of an arbitrarily large distributed system. We show that the implementation of the model is hampered by the limitations in display technology, resulting in the accurate display of a subset of the events of the execution.

Chapter 4

An Accurate Technique

The concurrency map uses vertical positions to identify the relationships among blocks of events. Concurrent relationships are shown by placing blocks such that their rows overlap, and causal relationships are shown by non-overlapping block placement. Horizontal separation only identifies the process that executed the block of events. Therefore, a single dimension is used to display both the causality and concurrency of a distributed system. As proven in the preceding chapter, this technique cannot accurately depict most execution scenarios. In this chapter we develop a technique that will accurately display both the causality and concurrency of a distributed system. The technique does so by restricting the number of processes that can be simultaneously displayed.

4.1 Display Coordinates

Assume that we are given a trace of the execution of the distributed system. The traced events need not contain vector times since the algorithm will reconstruct them. Also assume that the system uses lossless, point-to-point FIFO communication. That is, all messages sent from P_i to P_j arrive in the order they were sent.

An example four-process trace is given in table 4.1. The events are represented as either **S** (message transmission) or **R** (message receipt). Each event is followed by a number indicating the index of the communication partner process. For example, the first event of process P_0 is a message transmission (send) to process P_1. Since local computation events affect neither concurrency nor causality, they can be ignored without loss of generality.

P_0	P_1	P_2	P_3
S 1	S 0	S 0	R 2
S 3	R 0	S 3	R 0
S 2	R 2	R 0	S 2
R 2	R 0	S 1	S 0
S 1	S 3	R 3	R 1
R 1	S 2	R 1	R 2
R 3	S 0	S 3	S 1
R 1	S 0	R 0	R 1
S 2	R 3	R 0	
S 2	R 2	S 1	
R 2	S 3	S 0	
S 1	R 0		
R 1			

Table 4.1: Four process trace.

The time-space diagram from the example trace file is shown in figure 4.1. In the time-space diagram, the dotted-line labeled $C(v)^{\perp}$ indicates the beginning of CR_v, the concurrent region of event v, and follows the last event in P_j, for all j, that causally precedes v. As discussed in chapter 2, we assume that the execution of each process begins with an initial event. These events are not shown in the figure, but have vector times where all components are zero. Vector time comparison shows that any initial event causally precedes any non-initial event so that an event exists in each process that causally precedes v.

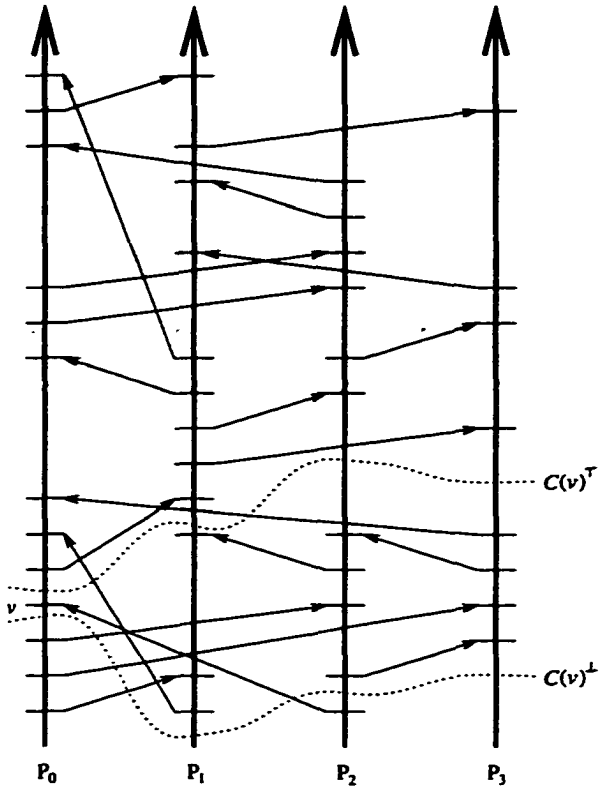


Figure 4.1: Time space diagram for example trace.

Definition 4.1 *For some process P_j, the cut $C(v)^{\perp}$ intersects P_j between events v_j and v'_j if and only if*

1. $v_j \rightarrow v'_j$,
2. $v_j \rightarrow v$,
3. $v'_j \not\rightarrow v$, *and*
4. $\nexists v''_j \in P_j : v_j \rightarrow v''_j \rightarrow v'_j$.

The dotted-line labeled $C(v)^{\top}$ indicates an alternate cut defined with respect to event v. This cut marks the end of CR_v and precedes the first event of process P_j, for all j, that causally follows v. Events between $C(v)^{\perp}$ and $C(v)^{\top}$ are in CR_v. We assume that each process ends its execution with a terminal event that has a vector time with all components

set to infinity. Vector time comparison shows that any terminal event causally follows all non-terminal events so that an event exists in each process that causally follows v.

Definition 4.2 *For some process P_j, the cut $C(v)^{\top}$ intersects P_j between events v_j and v'_j if and only if*

1. $v_j \rightarrow v'_j$,
2. $v \not\rightarrow v_j$,
3. $v \rightarrow v'_j$, *and*
4. $\nexists v''_j \in P_j : v_j \rightarrow v''_j \rightarrow v'_j$.

The system cuts $C(v)^{\perp}$ and $C(v)^{\top}$ partition the events of the system into three sets with respect to event v. Events that precede $C(v)^{\perp}$ are referred to as the "*before*" events (set B in chapter 3) and are those events that *happen before* v. Events that follow $C(v)^{\top}$ are in the "*after*" set (set A in chapter 3) and are the events that *happen after* v. The events that fall between $C(v)^{\perp}$ and $C(v)^{\top}$ are in the "*concurrent*" set (set C in chapter 3) and are the events that are concurrent to v. Note that the symmetric property of concurrency insures us that if event v' is in the concurrent set of event v, then event v will also be in the concurrent set of v'.

We represent each cut as a vector of real numbers, derived directly from system vector time. Suppose that $C(v)^{\perp}$ immediately follows event v' in process P_j. We assign to $C(v)^{\perp}[j]$ the value of the j^{th} component of the vector time of v' after it has been increased by 0.5. The increase is to clarify the display. Event v_i^k will be shown to begin at $k - 0.5$ on the P_i-axis. Definition 4.3 formalizes the assignment of values to components of $C(v)^{\perp}$.

Definition 4.3 *The system cut $C(v)^{\perp}$ is represented as a vector of real numbers where, for all j,*

$$C(v)^{\perp}[j] = \tau(v')[j] + 0.5$$

if $\mathcal{P}(v') = P_j$ and $C(v)^{\perp}$ immediately follows v' in P_j. Note that $C(v)^{\perp}[i] = \tau(v)[i] - 0.5$ since v' immediately precedes v in P_i.

In a similar manner, the components of $C(v)^{\top}$ are computed from the vector times of the events immediately following $C(v)^{\top}$. Suppose that $C(v)^{\top}$ immediately precedes event v' in process P_j. We assign to $C(v)^{\top}[j]$ the value of the j^{th} component of the vector time of v' after it has been decreased by 0.5. Since the cut precedes event v', we must show that the concurrent region does not include v'. Event v_i^k will be shown to end at $k + 0.5$ on the P_i-axis. Taken together with the previous definition, we see that all events executed in a single process have non-overlapping regions.

Definition 4.4 *The system cut $C(v)^{\top}$ is represented as a vector of real numbers where, for all j,*

$$C(v)^{\top}[j] = \tau(v')[j] - 0.5$$

if $\mathcal{P}(v') = P_j$ and $C(v)^{\top}$ immediately precedes v' in P_j.

Note that it is possible for the cuts $C(v)^{\perp}$ and $C(v)^{\top}$ to intersect a process at the same point. This will be the case if no events in that process are concurrent to v. Since the difference in the local components of the vector times of events is one, and we alter the vector

times by only 0.5 in computing the components of $C(v)^{\perp}$ and $C(v)^{\top}$, we are assured that $C(v)^{\top} \geq C(v)^{\perp}$ for all v. If no events in P_j are concurrent to v, then $C(v)^{\top}[j] = C(v)^{\perp}[j]$.

These two vectors are used to plot the events on a two-dimensional grid that preserves both the causal and concurrent relationships between displayed events. Each event will be represented as a rectangle with coordinates derived directly from the concurrent regions of the event. Two processes will be chosen and all events from only those processes will be displayed.

Definition 4.5 *The* display *of event v, $\mathcal{D}(v)$, given that $\mathcal{P}(v) = P_i$ and P_j is the second process chosen for presentation, is defined by the diagonal vertices $(C(v)^{\perp}[i], C(v)^{\perp}[j])$ and $(C(v)^{\top}[i], C(v)^{\top}[j])$.*

We can compare the values assigned to the displays to determine whether or not the graphical representations of the displays will intersect. If the maximum value in any dimension of one event's display is less than or equal to the minimum value of another event's display in the same dimension, then the events' displays are disjoint. For example, consider the two events, v from P_i and v' from P_j, shown in figure 4.2. The values of the displays are as follows.

$$\begin{aligned} \mathcal{D}(v) &= \{(1.5, 2.5), (3.5, 6.5)\} \\ \mathcal{D}(v') &= \{(4.5, 3.5), (7.5, 7.5)\} \end{aligned}$$

In the P_i-dimension the maximum of event v is 3.5 and the minimum of v' is 4.5. Since the maximum is less than the minimum, we know that the displays do not intersect. We do not consider figures that share a common border to be intersecting.

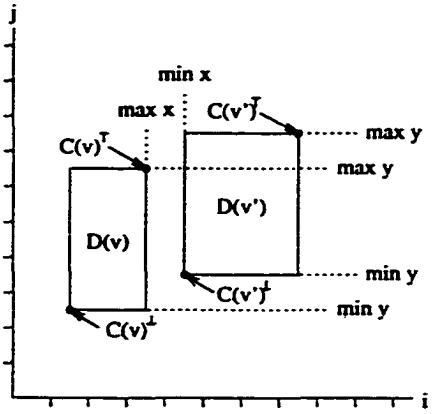


Figure 4.2: Display of events v and v'

We now show that the displays constructed as given above accurately display both the causality and concurrency of the distributed system. We begin by proving that causality is correctly characterized. That is, that the representations of two events are disjoint only if the events are causally related. We use the notation $\mathcal{D}(v) \cap \mathcal{D}(v')$ to indicate the graphical intersection of the representations of events v and v'.

Theorem 4.1 *Events v and v' are causally related, $v \not\parallel v'$, if and only if $\mathcal{D}(v) \cap \mathcal{D}(v') = \emptyset$.*

Proof: Let $v \in E_i$ and $v' \in E_j$.

We first show that $v \not\parallel v' \Rightarrow \mathcal{D}(v) \cap \mathcal{D}(v') = \emptyset$. Assume that $v \rightarrow v'$.

v precedes $C(v')^{\perp}$	$v \rightarrow v'$ and def 4.1	(4.1)
$C(v')^{\perp}[i] \geq \tau(v)[i] + 0.5$	(4.1) and def 4.3	(4.2)
$C(v')^{\perp}[j] \geq \tau(v')[j] - 0.5$	Def 4.1	(4.3)

v' follows $C(v)^\top$	$v \rightarrow v'$ and def 4.2	(4.4)
$C(v)^\top[j] \leq \tau(v')[i] - 0.5$	(4.4) and def 4.4	(4.5)
$C(v)^\top[i] = \tau(v)[i] + 0.5$	Def 4.2	(4.6)
$C(v')^\perp[j] \geq C(v)^\top[j]$	(4.3) and (4.5)	(4.7)
$C(v')^\perp[i] \geq C(v)^\top[i]$	(4.2) and (4.6)	(4.8)
$\mathcal{D}(v) \cap \mathcal{D}(v') = \emptyset$	(4.7) and (4.8)	(4.9)

We next show that $\mathcal{D}(v) \cap \mathcal{D}(v') = \emptyset \Rightarrow v \not\| v'$. Since the displays are disjoint, either $C(v')^\perp[i] \geq C(v)^\top[i]$ or $C(v)^\perp[i] \geq C(v')^\top[i]$. Assume that $C(v')^\perp[i] \geq C(v)^\top[i]$. The other case is proven similarly.

$C(v')^\perp[i] \geq C(v)^\top[i]$	$\mathcal{D}(v) \cap \mathcal{D}(v') = \emptyset$ and def 4.5	(4.10)
$C(v)^\top[i] = \tau(v)[i] + 0.5$	Def 4.2	(4.11)
$C(v')^\perp[i] \geq \tau(v)[i] + 0.5$	(4.10) and (4.11)	(4.12)
v precedes $C(v')^\perp$	(4.12) and def 4.3	(4.13)
$v \rightarrow v'$	(4.13) and def 4.1	(4.14)

Therefore, $v \not\| v'$ if and only if $\mathcal{D}(v) \cap \mathcal{D}(v') = \emptyset$. ■

Corollary 4.1 shows that the displays of two events intersect only if the events are concurrent. It is a direct application of the preceding theorem and the definition of concurrency.

Corollary 4.1 *Event $v \| v'$ if and only if $\mathcal{D}(v) \cap \mathcal{D}(v') \neq \emptyset$.*

We continue by proving that the direction of causality is correctly shown. Given two causally related events, v and v', where $v \rightarrow v'$, it must be the case that the display of v lies closer to the origin than does the display of v'.

Theorem 4.2 *Given two causally related events, v and v', $\mathcal{D}(v)$ is closer to the origin than $\mathcal{D}(v')$ if and only if $v \rightarrow v'$.*

Proof: The distance from $\mathcal{D}(v)$ to the origin is computed as follows.

$$\Delta_v = \sqrt{C(v)^{\perp}[i]^2 + C(v)^{\perp}[j]^2} \tag{4.15}$$

We first show that $v \rightarrow v' \Rightarrow \Delta_v < \Delta_{v'}$.

$C(v)^{\perp}[i] = \tau(v)[i] - 0.5$	Def 4.3	(4.16)
$C(v')^{\perp}[i] \geq \tau(v)[i] + 0.5$	$v \rightarrow v'$ and def 4.3	(4.17)
$C(v')^{\perp}[i] \geq C(v)^{\perp}[i] + 1$	(4.16) and (4.17)	(4.18)
$C(v')^{\perp}[i] > C(v)^{\perp}[i]$	(4.18)	(4.19)
$C(v)^{\top}[j] \leq \tau(v')[j] - 0.5$	$v \rightarrow v'$ and def 4.4	(4.20)
$C(v)^{\perp}[j] \leq \tau(v')[j] - 0.5$	$C(v)^{\perp}[j] \leq C(v)^{\top}[j]$ and (4.20)	(4.21)
$C(v')^{\perp}[j] = \tau(v)[j] + 0.5$	$v \rightarrow v'$ and def 4.3	(4.22)
$C(v)^{\perp}[j] \leq C(v')^{\perp}[j] - 1$	(4.21) and (4.22)	(4.23)
$C(v)^{\perp}[j] < C(v')^{\perp}[j]$	(4.23)	(4.24)
$\Delta_v < \Delta_{v'}$	(4.19) and (4.24)	(4.25)

We next show that $\Delta_v < \Delta_{v'} \Rightarrow v \rightarrow v'$. There are two possibilities for the inequality.

$$C(v)^{\perp}[i] < C(v')^{\perp}[i] \vee C(v)^{\perp}[j] < C(v')^{\perp}[j] \qquad \Delta_v < \Delta_{v'} \text{ and (4.15)} \qquad (4.26)$$

We first assume that $C(v)^{\perp}[i] < C(v')^{\perp}[i]$.

$C(v)^{\perp}[i] = \tau(v)[i] + 0.5$	Def 4.1	(4.27)
$\tau(v)[i] + 0.5 < C(v')^{\perp}[i]$	(4.26) and (4.27)	(4.28)
$C(v')^{\perp}$ follows v in P_i	(4.28) and def (4.3)	(4.29)
$v \rightarrow v'$	(4.29) and def 4.1	(4.30)

We now assume that $C(v)^{\perp}[j] < C(v')^{\perp}[j]$.

$C(v')^{\perp}[j] = \tau(v')[j] + 0.5$	Def 4.1	(4.31)
$C(v)^{\perp}[j] < \tau(v')[j] + 0.5$	(4.31) and (4.27)	(4.32)
$C(v)^{\perp}$ precedes v' in P_j	(4.32) and def (4.3)	(4.33)
$v' \not\rightarrow v$	(4.33) and def 4.1	(4.34)
$v \rightarrow v'$	$v \nparallel v'$ and (4.34)	(4.35)

Therefore, the display of v must be closer to the origin than the display of v' if and only if $v \rightarrow v'$. ■

We can select any two processes and accurately display the relationships between the events of those processes. Suppose we choose to show the relationships between events of

event	$C(v)^{\perp}$	$C(v)^{\top}$
v_0^1	[0.50, 0.50, 0.50, 0.50]	[1.50, 1.50, 2.50, 1.50]
v_0^2	[1.50, 0.50, 0.50, 0.50]	[2.50, 2.50, 2.50, 1.50]
v_0^3	[2.50, 0.50, 0.50, 0.50]	[3.50, 2.50, 2.50, 4.50]
v_0^4	[3.50, 0.50, 1.50, 0.50]	[4.50, 3.50, 5.50, 4.50]
v_3^1	[0.50, 0.50, 2.50, 0.50]	[6.50, 8.50, 4.50, 1.50]
v_3^2	[2.50, 0.50, 2.50, 1.50]	[6.50, 8.50, 4.50, 2.50]
v_3^3	[2.50, 0.50, 2.50, 2.50]	[6.50, 8.50, 4.50, 3.50]
v_3^4	[2.50, 0.50, 2.50, 3.50]	[6.50, 8.50, 7.50, 4.50]

Table 4.2: Vectors and display coordinates.

P_0 and P_3. We construct the grid by assigning one process to each axis. Table 4.2 shows the values of vectors $C(v)^{\perp}$ and $C(v)^{\top}$ for the first four events of processes P_0 and P_3 of our example. We can derive all other coordinates from these two coordinates due to the regularity of the figure. The i-axis is used to show possible event occurrence with respect to process P_i.

In the next section we describe an algorithm that will compute both $C(v)^{\perp}$ and $C(v)^{\top}$ for all events in the system. The internal structure of the algorithm will maintain vector times for the two cuts. Only when the final values are output do we increase or decrease the integers by 0.5.

The algorithm can either be executed in conjunction with the underlying computation or can be performed on trace files. In vitro execution does not require additional messages until the computation completes but does increase the size of most messages. The time complexity of the algorithm is linear in the number of events and is scalable to an arbitrary number of processes. Space complexity of the basic algorithm is exponential in the number

of messages in the worst case but can be bounded by inserting additional messages into the system's execution.

4.2 The Algorithm

For each message in the system, we need to know the time that each other process becomes causally aware of that message. When a message arrives, the current time as well as the message originator (transmitting process) are noted.

Later, when a message is sent back to the originator, the noted values are appended. We will refer to the appended values as an *acknowledgment* even though they do not fit the traditional definition. The appended values perform two functions. First, they inform the originating process of the time the receiving process became aware of this and all causally preceding messages. This time is used to update $C(v)^{\top}$. Second, by forwarding the values to other processes, the transitivity of causality is preserved.

For example, suppose that P_i receives messages 1 and then 3 from P_j while message 2 was sent to another process. When message 3 arrives at P_i, the causal information of message 1 and 2 also arrives regardless of whether the messages are physically at P_i. The acknowledgment that will eventually be sent back to P_j will inform P_j of the time when P_i became aware of messages 3 and 2. Our algorithm maintains $C(v)^{\top}$ to indicate the time that a process became causally aware of a message. Note that the maintained values are not the same as the resulting values. When the acknowledgment arrives as P_j, it is used to update the $C(v)^{\top}$ vectors of the events preceding the transmission of message 3, which will include the transmission of message 2.

Each event is represented as a four-tuple, $(type, partner, causalVT, concurrentVT)$. The *type* of each event is either *send* or *recv* and the *partner* is a process identifier that indicates which other process is involved in the communication. A vector time is maintained in $causalVT$ which represents $C(v)^{\perp}$ and is initialized to the 0 vector to indicate no events from any process causally precede this event. An additional vector is kept in $concurrentVT$ which contains the known components of $C(v)^{\top}$. The values of this vector's components are initialized to ∞ indicating undetermined values.

In addition to the underlying computational content which is ignored in the post mortem algorithm, each message will have data appended to it. Messages will be of the format $(vectorTime, appendList)$ where $vectorTime$ is the vector time of the transmitting event. The $appendList$ is a list of acknowledgments, each an originator/vector time pair, that are to be returned or forwarded to the destination process. These times represent the causal dependencies carried by this message.

Processes are slightly more complex than are events or messages. Each processes is represented as a six-tuple, $(PID, vectorTime, eventList, inputList, outputList, lastAcked)$. The PID is the processes identifier and $vectorTime$ is the current vector time of the process. This time will be used to stamp events as they are processed. All events are inserted into the $eventList$ at the beginning of the algorithm and are updated in place. Messages sent to P_i are inserted into the corresponding $inputList$ by the sending process. Messages contained in $P_i.inputList_j$ are those messages send from P_j to P_i but not yet consumed. Data to be appended on the next message sent to P_j will be found in $P_i.outputList_j$. Stored in $P_i.lastAcked_j$ is the highest indexed event of P_j with an acknowledged inserted into $P_i.inputList_j$.

```
foreach process P_i do
    while traceFile_i not empty do                    /* iterate of over entire file */
        read event from traceFile                     /* event type and partner processes ID */
        foreach process P_j do                        /* initialize event vectors */
            event.causalVT_j = 0                      /* no causally preceding event */
            event.concurrentVT_j = ∞                  /* all events assumed concurrent */
        od. /* foreach P_j */
        eventList_i = eventList_i ∪ {event}           /* events are kept in order */
    od  /* while not empty */
    foreach process P_j do                            /* create empty lists */
        P_i.inputList_j = φ                           /* no incoming messages */
        P_i.outputList_j = φ                          /* no causal links to forward */
        P_i.vectorTime_j = 0                          /* process vector time */
        P_i.lastAcked_j = 0                           /* last event acknowledged */
    od  /* foreach P_j */
od  /* foreach P_i */
```

Algorithm 4.1: Algorithm initialization.

Algorithm initialization, as shown in algorithm 4.1, opens and reads events from the trace files. As each event is read, it is used to construct a record with two vector times initialized to 0 and ∞ as described above. The record is then inserted into the *eventList*. Also in the initialization phase, each process creates N *inputLists* and N *outputLists*, all empty. A vector time is kept for each process which represents the current vector time and is used to stamp events as they are processed. This vector time and the *lastAcked* vector are initialized to all zero values.

```
P_i.vectorTime_i ++                                   /* a new event ID */
event.causalTime = P_i.vectorTime                     /* stamp new time on event */
event.concurrentTime_i = P_i.vectorTime_i + 1         /* only concurrent to itself in P_i */
message = (P_i.vectorTime, P_i.outputList_j)          /* contains no data if post mortem */
P_j.inputList_i = P_j.inputList_i ∪ {message}         /* message transmission */
P_i.lastAcked_i = P_i.vectorTime_i                    /* no need to acknowledge this message */
P_i.outputList_j = φ                                  /* new, empty list */
```

Algorithm 4.2: Processing a send event.

After initialization, the algorithm cycles through the processes from P_0 to P_{N-1} until an entire pass is made without consuming an event. This indicates that either all events have been processed or deadlock has occurred. During each pass, the events of process P_i are examined to determine their type. If the event is determined to be a send event of a message from P_i to P_j, we follow algorithm 4.2 that first increments the i^{th} component of the process' vector time. The updated vector time is then used to set the event's causal and concurrent vectors. A new message is created which contains the updated vector time and the possibly empty output list for P_j. This message is appended to the input list at P_j for future consumption. We then set the i^{th} component of the *lastAcked* vector to the index of this event since we are already aware of its time of occurrence in P_i. Finally, a new, empty output list is created and we continue with the next event in P_i.

If the event is a receive, we check to insure that a message is ready to be consumed. If no message is in the input list from the partner (sending) process, the receive event must wait. We simulate this wait by continuing with the next process and retrying the event during the next round. Algorithm 4.3 shows this test.

```
if P_i.inputList_j.isEmpty                    /* blocking receive, no message waiting */
    continue with next process                        /* try receive again next round */
fi  /* P_i.inputList_j = φ */
```

Algorithm 4.3: Insure a message can be received from P_j at P_i.

If a message is available in the input list from P_j, it is removed from the list and processed in two parts. We first consider the message proper as shown in algorithm 4.4. The vector time of the process is incremented to indicate the occurrence of an event and then set to the

component-wise maximum of the new vector time and the time stamp of the message. We now have an updated process vector time that accurately reflects the causal dependencies of this event. We set the causal vector time of the event to the updated process vector time. To show that this event is only concurrent to itself in P_i, we set the i^{th} component of the concurrent vector time of the event to the i^{th} component of the process' vector time. If the message has not been acknowledged, a new message appendage is constructed with the identifier of the sending process and all components of the vector set to ∞ except the i^{th} and j^{th} components which are set equal to their associated components of the process' vector time. This new appendage is inserted into all output lists to indicate the receipt of the message. Since this message has now been acknowledged, we set the j^{th} component of the *lastAcked* vector to the j^{th} component of the process' vector time.

Processing the message content inserts information into the output lists to indicate the time of the receipt of that message. However, we have not only become aware of this message but all other messages that preceded it. The appended list is processed to update local event $C(v)^{\top}$ and forward the transitive arrivals to other processes.

As shown in algorithm 4.5, each appendage is processed in turn. We begin by extracting $origPID$, the identifier of the originating process. The originator may be any process in the system except P_j. If the vector time of the appendage has a larger value in the $origPID$ component than does the *lastAcked* vector, then the message has not been previously acknowledged. Note that it is not possible for the message to pass this test and to have originated at P_i. We update the last acknowledged vector and the i^{th} component of the appendage's vector time. The newly updated appendage must be forwarded to all processes that still have ∞ in their component of the appendage's vector time. The ∞ value indicates

```
msg = P_i.inputList_j.nextMessage          /* get next message from input queue */
P_i.vectorTime_i ++                                          /* a new event ID */
foreach process P_k do                                       /* update vector time */
    P_i.vectorTime_k = max(P_i.vectorTime_k, msg.vectorTime_k)   /* component-wise max */
od  /* foreach P_k */
event.causalTime = P_i.vectorTime                            /* stamp new time on event */
event.concurrentTime_i = P_i.vectorTime_i + 1                /* only concurrent to itself in P_i */
if msg.vectorTime_j > P_i.lastAcked_j                        /* receipt needs to be acked */
    P_i.lastAcked_j = P_i.vectorTime_j                       /* remember the acknowledgment */
    append.PID= j                                            /* P_j originated the message */
    foreach process P_k do                                   /* initialize appendage vector */
        append.vectorTime_k = ∞                              /* no processes aware of message */
    od  /* foreach P_k */
    append.vectorTime_i = P_i.vectorTime_i                   /* P_i is aware of message */
    append.vectorTime_j = P_i.vectorTime_j                   /* P_j is aware of message */
    foreach process P_k≠i do                                 /* add message to output lists */
        P_i.outputList_k = P_i.outputList_k ∪ {append}       /* forward to P_k */
    od  /* foreach P_k */
fi  /* msg.vectorTime_j > P_i.lastAcked_j */
```

Algorithm 4.4: Processing the message content.

that the message hasn't yet been seen by that process. It must also be sent to the originator of the message so that the concurrency time of the sending event can be updated.

Suppose, however, that the message originated at P_i. We must then use the information contained in the appendage to update $C(v)^\top$ of the sending event of the message. By extracting the i^{th} component of the attachment's vector time we can obtain the event number (index) of the send event being acknowledged. Beginning at this event and working backwards through the event list, we examine $C(v)^\top$ of each event. Each component of $C(v)^\top$ is compared to the corresponding component of the appendage. If the appendage component is smaller, we set the event's component to that value. When the event list is exhausted we stop processing. We may also stop processes when we check all components of

```
foreach append ∈ msg.appendList do          /* what other messages were transitively sent */
   origPID = append.PID                                  /* originating process identifier */
   if append.vectorTime_origPID > P_i.lastAcked_origPID   /* receipt needs acknowledgment */
      P_i.lastAcked_origPID = append.vectorTime_origPID     /* only acknowledge once */
      append.vectorTime_i = P_i.vectorTime_i                /* time seen at P_i */
      foreach process P_k≠i do
         if append.vectorTime_k = ∞                         /* P_k isn't aware */
            P_i.outputList_k = P_i.outputList_k ∪ {append}   /* forward to P_k */
         fi  /* append.vectorTime_k = ∞ */
      od  /* foreach P_k */
      P_i.outputList_origPID = P_i.outputList_origPID ∪ {append}   /* send ack to P_j */
   elif origPID= i                                       /* message originated at P_i */
      eventNum = append.vectorTime_i                     /* index of send event */
      changed = true                                     /* flag indicating changes are complete */
      while (eventNum > 0 and changed) do                /* no more events or no change */
         prevEvent = P_i.eventList_eventNum              /* get event from eventList */
         changed = false                                 /* toggled in loop if changes are made */
         foreach process P_k≠i do
            if prevEvent.concurrentTime_k > append.vectorTime_k   /* a new minimum */
               prevEvent.concurrentTime_k = append.vectorTime_k   /* new time */
               changed = true                                     /* a change was made */
            fi  /* prevEvent.concurrentTime_k > append.vectorTime_k */
         od  /* foreach P_k */
         eventNum - -                                    /* previous event in P_i */
      od  /* (eventNum > 0 and changed) */
   end if  /* origPID= i */
od  /* append ∈ msg.appendList */
```

Algorithm 4.5: Processing the appended information.

an event's $C(v)^{\mathsf{T}}$ and none are altered. This indicates that the concurrent region identified by the appendage does not include this or any previous events.

By updating $C(v)^{\mathsf{T}}$ of events in this manner, we are assured that no event will be visited more than $N-1$ times. When an event is changed, at least one component is updated to the correct value. While it is possible that some components will be changed again to reflect a different path through the causal chain, the one minimum will not be changed. If the

event is visited $N-1$ times, then all components will be correctly set and processing will not visit preceding events.

In addition, by forwarding appendages only to those processes that are not previously aware of the message, we reduce the size of each message. While it would be ideal to insure that we do not forward an indication of a message to a process that is already aware of that message, it is not possible to do so. We must instead be satisfied to forward appendages only to those processes that we do not know are already aware of the message.

Suppose for example that a message is sent from P_i to P_j. When that message is received, an acknowledgment appendage is inserted into the output lists destined for each processes except P_j. If a second message is sent from P_i to P_k, it will contain information regarding the first message. A third message sent this time from P_j to P_k will have an appendage noting the causal relation with the first message. Since P_k is already aware of the first message, the appendage is redundant. It will, however, be sent by P_j and discarded at P_k.

When iteration stops, either because all events have been processed or deadlock has occurred, we send one final round of messages as shown in algorithm 4.6. These messages have the sole purpose of transferring the output lists of each process to their intended destination. Since they are not sent until the underlying computation has terminated, they impose little overhead on the distributed system. Each process will send a message containing only the current vector time and the contents of the output list to each of the other processes. When this portion of the algorithm completes, each process knows whether or not each message it sent arrived at each other process. Furthermore, for each message that did arrive either directly or transitively, the time of the arrival is known.

```
P_i.vectorTime_i ++                                  /* one greater than last event ID */
message = (P_i.vectorTime, P_i.outputList_j)         /* contains no data */
foreach process P_{j≠i} do                           /* message to each other process */
    P_j.inputList_i = P_j.inputList_i ∪ {message}    /* message transmission */
    P_i.outputList_j = φ                             /* new, empty list */
od  /* foreach P_j */
```

Algorithm 4.6: Transmitting the output lists.

Algorithm 4.7 shows the steps needed to process the incoming, final round messages at P_i. As messages are received, each appendage is checked to insure it represents a message originating at P_i. Only acknowledgments of messages originating at P_i are of use. Others are discarded without being processed. Those not discarded are used to set the components of the concurrent vector times of the send event they represent and events preceding that send event. Just as in the previous receive algorithm, only when a component will be lessened by the change is the alteration performed.

When all appendages of a message have been processed, the vector time of the message is used to compute a value for infinity. If x is the index of the last event executed in process P_j, then the j^{th} component of the message vector time will be $x+1$. Since the value ∞ represents the end of time for the process, we can replace it with the new value of the end of time without compromising the algorithm. The replacement also provides a usable value of $C(e)^{\top}$ for the display of the event. Each event in the event list of P_i is checked to see if the j^{th} component of its concurrency time is greater then the j^{th} component of the message's vector time. If it is, the vector time component is copied into the concurrency time component.

```
foreach process P_{j≠i} do                                   /* iterate over all processes */
   msg = P_i.inputList_j.nextMessage                         /* get message from input queue */
   foreach append ∈ msg.appendList do                        /* only process the appendages */
      if append.PID= i                                       /* only process if originated at P_i */
         eventNum = append.vectorTime_i                      /* index of send event */
         changed = true                                      /* flag indicating changes are complete */
         while (eventNum > 0 and changed) do                 /* no more events or no change */
            prevEvent = P_i.eventList_{eventNum}             /* get event from eventList */
            changed = false                                  /* toggled in loop if changes are made */
            foreach process P_{k≠i} do
               if prevEvent.concurrentTime_k > append.vectorTime_k   /* new min */
                  prevEvent.concurrentTime_k = append.vectorTime_k   /* new time */
                  changed = true                             /* change made */
               fi  /* prevEvent.concurrentTime_k > append.vectorTime_k */
            od  /* foreach P_k */
            eventNum - -                                     /* previous event in P_i */
         od  /* (eventNum > 0 and changed) */
      fi  /* append.PID= i */
   od  /* append ∈ msg.appendList */
   eventNum= P_i.numEvents                                   /* look for infinity */
   changed = true                                            /* toggled in loop if changes are made */
   while (eventNum > 0 and changed) do                       /* no more events or no change */
      prevEvent = P_i.eventList_{eventNum}                   /* get event from eventList */
      changed = false                                        /* toggled in loop if changes are made */
      if prevEvent.concurrentTime_j > append.vectorTime_j    /* a tight infinity */
         prevEvent.concurrentTime_j = append.vectorTime_j    /* new infinity */
         changed = true                                      /* change made */
      fi  /* prevEvent.concurrentTime_j > append.vectorTime_j */
      eventNum - -                                           /* previous event in P_i */
   od  /* eventNum > 0 and changed */
od  /* process P_{j≠i} */
```

Algorithm 4.7: Processing the final appendages.

When this part of the algorithm completes, all events in P_i will have valid values for all components of their causal and concurrent vector times. The next section steps through the first round of processing, listing the values of each variable.

4.3 An Example

To demonstrate the algorithm we will step through the first round of execution using the four-process trace of table 4.1 as input. The first three events in P_0 are message transmissions and can be executed without delay. Each event causes the process vector time of P_0 to be incremented and stamped onto the event. This vector time and the empty output lists are used to construct messages. We then append these messages to the input lists of the recipient process. Since the fourth event is a receive event and no message is waiting in the input list of P_0, we begin processing P_1. When P_0 has finished this round, its process vector time is $[3,0,0,0]$ and each of the other processes have a message from P_0 in their input list.

Process P_1 executes the first event in its event list, a transmission to P_0, in the same manner as did process P_0. The process vector time is incremented and stamped onto the event. A message is then created with the vector time of the event and an empty output list, and added to the the input list of P_0.

Next to be processed is a message receipt from P_0. Since an incoming message is ready to be consumed, that is, $P_1.inputList_0$ is not empty, the event is processed. The process's vector time is incremented and then updated to reflect the causality transferred through the message. Since no messages from P_0 have previously been acknowledged, a new appendage is created and added to the output list destined for each other processes. This appendage has the structure $(0,[1,2,\infty,\infty])$ indicating that the message to be acknowledged originated in process P_0 as event number 1 and was received in process P_1 by event number 2. Values of infinity for the other two components indicate that the message has not yet been received

at P_2 or P_3. No more events in P_1 can be processed since the next event is a receipt from P_2 and no message is currently waiting to be received.

Processing the first three events in P_2 is similar to that already described. The first two events are message transmissions to P_0 and then P_3 which increment the process vector time and include empty appendages with the messages. The third event, a receipt from P_0, updates the process vector time and inserts message acknowledgments $(0, [3, \infty, 3, \infty])$ into the output lists for each of the other processes. When the fourth event, a transmission to P_1, is processed, the output list destined for P_1 contains the acknowledgment of the receipt from P_0. When the message with the appendage arrives at P_1, it will not only carry the causal information about the message but also causal information about previous messages. Processing cannot continue in this process since the next event is a message receipt and the message is not in the input list.

Receipts from P_2 and then P_0 are processed by P_3. Appendages $(2, [\infty, \infty, 2, 1])$ and $(0, [2, \infty, \infty, 2])$ are added to the output lists and included in messages sent to P_2 and P_0. The message from P_1 is not ready to be received so processing at P_3 stops, ending the first round of processing.

At the end of round 1, the processes' structures contain the information shown in tables 4.3, 4.4, 4.5 and 4.6. The two vectors shown with each event v are $\tau(v)$ and $C(v)^{\mathsf{T}}$. The text string `INF` indicates a values of ∞ and messages without appendages are marked with `[NULL]`.

```
Process P0  (13 events)
:: vector time [    3,    0,    0,    0]
:: lastAcked [    3,    0,    0,    0]
:: current event :: 4 of 13
:: Event List ::
   S  1 [    1,    0,    0,    0] [    2,  INF,  INF,  INF]
   S  3 [    2,    0,    0,    0] [    3,  INF,  INF,  INF]
   S  2 [    3,    0,    0,    0] [    4,  INF,  INF,  INF]
   R  2 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   S  1 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   R  1 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   R  3 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   R  1 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   S  2 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   S  2 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   R  2 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   S  1 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   R  1 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
:: Output List (appendages) ::
:: P0
:: P1
:: P2
:: P3
:: Input List (messages) ::
   from P1  [    0,    1,    0,    0]
      [NULL]
   from P2  [    0,    0,    1,    0]
      [NULL]
   from P3  [    2,    0,    2,    4]
      (2, [  INF,  INF,    2,    1])(0, [    2,  INF,  INF,    2])
```

Table 4.3: P_0 after round 1.

```
Process P1  (12 events)
:: vector time [    1,    2,    0,    0]
:: lastAcked [    1,    1,    0,    0]
:: current event :: 3 of 12
:: Event List ::
   S  0 [    0,    1,    0,    0] [  INF,    2,  INF,  INF]
   R  0 [    1,    2,    0,    0] [  INF,    3,  INF,  INF]
   R  2 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   R  0 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   S  3 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   S  2 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   S  0 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   S  0 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   R  3 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   R  2 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   S  3 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   R  0 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
:: Output List (appendages) ::
:: P0
   (0, [    1,    2,  INF,  INF])
:: P1
:: P2
   (0, [    1,    2,  INF,  INF])
:: P3
   (0, [    1,    2,  INF,  INF])
:: Input List (messages) ::
   from P2  [    3,    0,    4,    0]
      (0, [    3,  INF,    3,  INF])
```

Table 4.4: P_1 after round 1.

```
Process P2  (11 events)
:: vector time [    3,    0,    4,    0]
:: lastAcked [    3,    0,    4,    0]
:: current event :: 5 of 11
:: Event List ::
   S  0 [    0,    0,    1,    0] [  INF,  INF,    2,  INF]
   S  3 [    0,    0,    2,    0] [  INF,  INF,    3,  INF]
   R  0 [    3,    0,    3,    0] [  INF,  INF,    4,  INF]
   S  1 [    3,    0,    4,    0] [  INF,  INF,    5,  INF]
   R  3 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   R  1 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   S  3 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   R  0 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   R  0 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   S  1 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   S  0 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
:: Output List (appendages) ::
:: P0
   (0, [    3,  INF,    3,  INF])
:: P1
:: P2
:: P3
   (0, [    3,  INF,    3,  INF])
:: Input List (messages) ::
   from P3  [    2,    0,    2,    3]
      (2, [  INF,  INF,    2,    1])(0, [    2,  INF,  INF,    2])
```

Table 4.5: P_2 after round 1.

```
Process P3  (8 events)
:: vector time [    2,    0,    2,    4]
:: lastAcked [    2,    0,    2,    4]
:: current event :: 5 of 8
:: Event List ::
   R  2 [    0,    0,    2,    1] [  INF,  INF,  INF,    2]
   R  0 [    2,    0,    2,    2] [  INF,  INF,  INF,    3]
   S  2 [    2,    0,    2,    3] [  INF,  INF,  INF,    4]
   S  0 [    2,    0,    2,    4] [  INF,  INF,  INF,    5]
   R  1 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   R  2 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   S  1 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
   R  1 [    0,    0,    0,    0] [  INF,  INF,  INF,  INF]
:: Output List (appendages) ::
:: P0
:: P1
   (2, [  INF,  INF,    2,    1])
   (0, [    2,  INF,  INF,    2])
:: P2
:: P3
:: Input List (messages) ::
```

Table 4.6: P_3 after round 1.

Each round processes events until all events have been consumed. The fifth round does not consume any events and the main processing loop exits. At this point, each process creates a final message for each other process. These messages contain the appendages remaining in the output lists and are inserted into the input lists of the other process. After the messages have been sent, each process reads the messages and processes the appendages to update the concurrency time vector of its events. After processing of the final messages has completed, the structures of the processes contain the information shown in tables 4.7, 4.8, 4.9 and 4.10.

```
Process P0  (13 events)
:: vector time [   14,    8,   11,    4]
:: lastAcked [   12,    8,   11,    4]
:: current event :: 14 of 13
:: Event List ::
   S  1 [    1,    0,    0,    0] [    2,    2,    3,    2]
   S  3 [    2,    0,    0,    0] [    3,    3,    3,    2]
   S  2 [    3,    0,    0,    0] [    4,    3,    3,    5]
   R  2 [    4,    0,    1,    0] [    5,    4,    6,    5]
   S  1 [    5,    0,    1,    0] [    6,    4,    6,    5]
   R  1 [    6,    1,    1,    0] [    7,   10,    8,    8]
   R  3 [    7,    1,    2,    4] [    8,   10,    8,    8]
   R  1 [    8,    7,    4,    4] [    9,   10,    8,    8]
   S  2 [    9,    7,    4,    4] [   10,   10,    8,    8]
   S  2 [   10,    7,    4,    4] [   11,   10,    9,    8]
   R  2 [   11,    7,   11,    4] [   12,   12,   12,    9]
   S  1 [   12,    7,   11,    4] [   13,   12,   12,    9]
   R  1 [   13,    8,   11,    4] [   14,   13,   12,    9]
:: Output List (appendages) ::
:: P0
:: P1
:: P2
:: P3
:: Input List (messages) ::
```

Table 4.7: P_0 after processing is complete.

```
Process P1  (12 events)
:: vector time [   12,   13,   11,    7]
:: lastAcked [   12,   11,   11,    7]
:: current event :: 13 of 12
:: Event List ::
   S  0 [    0,    1,    0,    0] [    6,    2,    6,    5]
   R  0 [    1,    2,    0,    0] [    8,    3,    6,    5]
   R  2 [    3,    3,    4,    0] [    8,    4,    6,    5]
   R  0 [    5,    4,    4,    0] [    8,    5,    6,    5]
   S  3 [    5,    5,    4,    0] [    8,    6,    6,    5]
   S  2 [    5,    6,    4,    0] [    8,    7,    6,    6]
   S  0 [    5,    7,    4,    0] [    8,    8,    8,    8]
   S  0 [    5,    8,    4,    0] [   13,    9,   12,    8]
   R  3 [    5,    9,    7,    7] [   14,   10,   12,    8]
   R  2 [   10,   10,   10,    7] [   14,   11,   12,    8]
   S  3 [   10,   11,   10,    7] [   14,   12,   12,    8]
   R  0 [   12,   12,   11,    7] [   14,   13,   12,    9]
:: Output List (appendages) ::
:: P0
:: P1
:: P2
:: P3
:: Input List (messages) ::
```

Table 4.8: P_1 after processing is complete.

```
Process P2  (11 events)
:: vector time [   10,    7,   12,    4]
:: lastAcked [   10,    7,   11,    4]
:: current event :: 12 of 11
:: Event List ::
   S  0 [    0,    0,    1,    0] [    4,    3,    2,    1]
   S  3 [    0,    0,    2,    0] [    7,    3,    3,    1]
   R  0 [    3,    0,    3,    0] [    8,    3,    4,    5]
   S  1 [    3,    0,    4,    0] [    8,    3,    5,    5]
   R  3 [    3,    0,    5,    3] [   11,    9,    6,    6]
   R  1 [    5,    6,    6,    3] [   11,    9,    7,    6]
   S  3 [    5,    6,    7,    3] [   11,    9,    8,    6]
   R  0 [    9,    7,    8,    4] [   11,   10,    9,    8]
   R  0 [   10,    7,    9,    4] [   11,   10,   10,    8]
   S  1 [   10,    7,   10,    4] [   11,   10,   11,    8]
   S  0 [   10,    7,   11,    4] [   11,   12,   12,    9]
:: Output List (appendages) ::
:: P0
:: P1
:: P2
:: P3
:: Input List (messages) ::
```

Table 4.9: P_2 after processing is complete.

```
Process P3  (8 events)
:: vector time [   10,   11,   10,    9]
:: lastAcked [   10,   11,   10,    7]
:: current event :: 9 of 8
:: Event List ::
   R  2 [    0,    0,    2,    1] [    7,    9,    5,    2]
   R  0 [    2,    0,    2,    2] [    7,    9,    5,    3]
   S  2 [    2,    0,    2,    3] [    7,    9,    5,    4]
   S  0 [    2,    0,    2,    4] [    7,    9,    8,    5]
   R  1 [    5,    5,    4,    5] [   14,    9,   12,    6]
   R  2 [    5,    6,    7,    6] [   14,    9,   12,    7]
   S  1 [    5,    6,    7,    7] [   14,    9,   12,    8]
   R  1 [   10,   11,   10,    8] [   14,   13,   12,    9]
:: Output List (appendages) ::
:: P0
:: P1
:: P2
:: P3
:: Input List (messages) ::
```

Table 4.10: P_3 after processing is complete.

This data is translated into four-dimensional coordinates and written to files for later use. We have written a simple program that constructs the display grid using the information stored in the output files. Figure 4.3 shows the window created by the program. The buttons along the bottom allow the selection of any two processes for display. Also along the bottom of the window is a button for including labels on each displayed event. If labels are desired, event v_i^k will be labeled on one side as $\mathbf{v}_{\mathbf{i}}^{\mathbf{k}}$.

Events from the lower numbered process, P_0 in the figure, are colored red, while the events from the higher numbered process, P_3 in the figure, are colored blue. The color assigned to a process is indicated by changing the color of the button's label at the bottom of the window. Event labels are drawn using the same color scheme as events. Also notice that the rectangles representing the events are slightly offset. This is not part of the

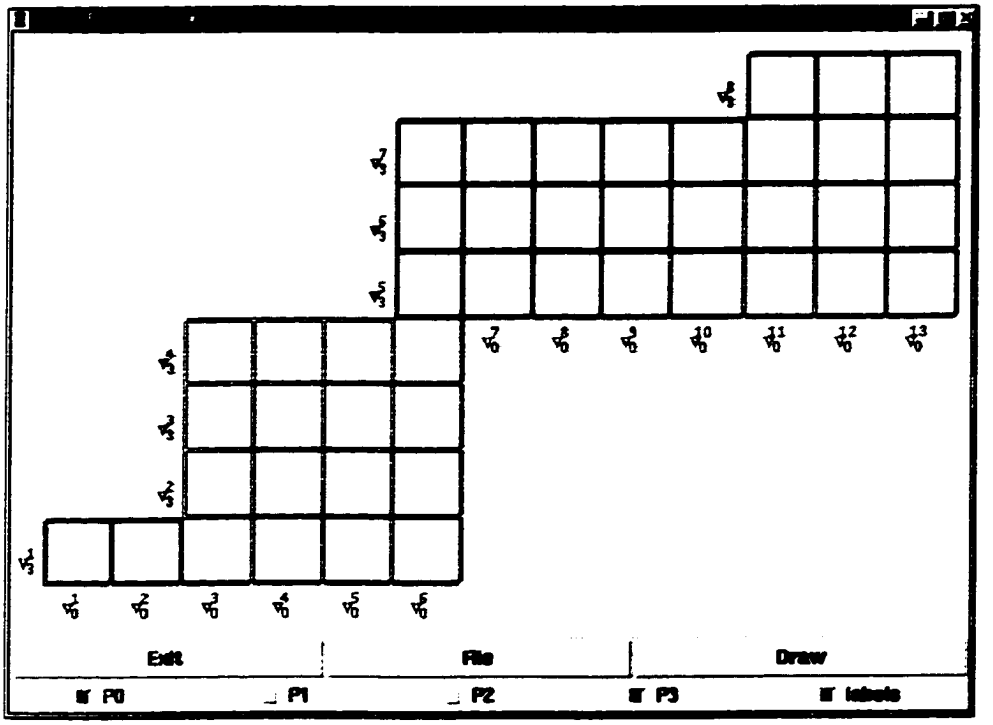


Figure 4.3: Snapshot of the `DisplayEvents` program.

theoretic result but an artifact of implementation. To prevent the first events drawn from being overwritten by the second events drawn, we offset each group. The events from P_0 are drawn one pixel to the left and one pixel down from the original location. Conversely, the events from P_3 are drawn one pixel up and one pixel to the right of the original location.

In the figure, we can see the relationship between all events of processes P_0 and P_3. Since the rectangular display of event v_3^2 intersects the rectangular display of event v_0^6, we know that the events are concurrent. If we examine the time-space diagram of figure 4.1, we see that the events are correctly displayed. Also shown in the picture is a causal relationship between events v_3^4 and v_0^7. The rectangular displays of these events do not intersect so the relationship is causal. Furthermore, since the display of v_3^4 is closer to the origin (in the

lower left corner) than the display of v_0^7, the direction of the causal link must be $v_3^4 \rightarrow v_0^7$. Again, by examining figure 4.1 we see that the causality is accurate.

4.4 Relaxation of Restrictions

We began this chapter by assuming that the communication paradigm underlying the system was lossless and point-to-point FIFO. We now consider the implications of relaxing these restrictions.

Consider two messages, m_1 and m_2, sent from process P_i to process P_j and received out of order. The current algorithm has no method of determining what message has arrived when a receive is executed. However, the addition of a sequence number in the trace file rectifies the problem. When each message is transmitted from P_i, the local component of the vector time, $\tau(m)_i$, is added to the information written to the trace file. When received at P_j, $\tau(m)_i$ is also appended to the traced information. This provides a means to display the receipt of the message that was transmitted. Instead of simply testing to see if a message is in the input queue, the test is now whether or not a particular message is in the input queue. If the particular message is located in the queue, it is consumed regardless of its queue position.

The effect on the display algorithm is negligible. The causal information contained in m_1 is also contained in m_2 by definition of causality. Upon arrival of m_2 the concurrent regions between P_i and P_j are formed. When m_1 arrives at some later time, it contains no useful information and is discarded.

Now consider the impact of a lost message on the display algorithm. If message m_1 is sent from P_i to P_j and never arrives, there are two possibilities. First, another message, m_2, sent after m_1 can arrive. In this circumstance, we are presented with exactly the situation just explained. Two messages are received out-of-order. The first to arrive is used to update the concurrent regions. Since message m_1 never arrives, we are spared the bookkeeping task of discarding it.

However, if message m_2 is never sent, or if all succeeding messages from P_j to P_i are also lost in transit, the concurrent regions between P_i and P_j are never updated. This is the correct result. The transmission of a message has no effect on causality or concurrency. Only when a message is received is the link formed. The display algorithm correctly surmises that the region of concurrency extends to the end of the process's life.

4.5 Evaluation

This technique has the merit of accurately displaying both causality and concurrency in a single static picture. However, several major downfalls are present.

- The model is limited to the simultaneous display of only two processes. It is difficult to extract much useful information from such a small subset of the system. This technique will fail in a complex system where the interactions of many processes are needed.

- The number of events is irrelevant only in the theoretic model. The complexity of the display increases with the number of events being simultaneously displayed. More

than a few events from each process create a representation from which little useful information can be extracted.

- While causal relationships are accurately shown, the messages creating those relationships are missing. It would be difficult for a software engineer to locate the source of an error based solely on the information presented in the figures.

We have shown in the previous chapter that the concurrency map, while being scalable and informative, does not accurately present the relationships of the system. In this chapter we developed a technique that accurately presents both causal and concurrency relationships at a cost of being unusable in practice. We believe that any technique showing causal and concurrent relationships using geometric means will fail to meet at least one of our criteria.

In the next chapter we reconsider the time-space diagram as the basis for a software engineering tool. We present a prototype and show how it can be used to quickly find and identify erroneous behavior in a complex distributed application.

Chapter 5

Distributed Trace Visualization Tool

An important result of this research was to develop a prototype display system called the Distributed Trace Visualization System, or DTVS[12]. In it we use a time-space diagram to display the relationships between events recorded during execution. Our implementation includes facilities for dynamic predicate definition, placement and evaluation as well as facilities for displaying a subset of the concurrency relationship.

The original implementation of DTVS displayed the execution of a distributed system using only synchronous message passing. It has been expanded to display the traces generated by programs using synchronous, asynchronous or broadcast communication paradigms. Currently, only programs written in C using a subset of the MPI[15, 16] libraries are traced.

5.1 Required Code Modifications

A header file is included in the source code to enable the tracing facilities of DTVS. This header contains the prototypes of several user callable functions. Preprocessor macros

replace selected MPI function calls with their equivalent DTVS function calls. Table 5.1 lists the MPI functions that are available with the DTVS tracing facilities. Other MPI constructs remain functional but are not included in the prototype.

Initialization and termination	
MPI_Init	Initialize the MPI execution environment
MPI_Finalize	Terminate the MPI execution environment
Message transmission	
MPI_Send	Transmission (nonblocking in DTVS)
MPI_Bsend	Transmission with user-specified buffering
MPI_Rsend	Ready transmission, blocking
MPI_Ssend	Synchronous transmission, blocking
MPI_Isend	Nonblocking transmission
Broadcast (send and receive)	
MPI_Bcast	Broadcast message transmission or receipt Choice depends on target process ID
Message receipt	
MPI_Recv	Blocking receive
MPI_Irecv	Nonblocking receive
MPI_Srecv	Synchronous receive, blocking
MPI_Wait	Returns when nonblocking receive has completed
MPI_Test	Returns true only if nonblocking receive has completed

Table 5.1: MPI functions available with DTVS.

For example, consider the following macro for `MPI_Send()`.

```
#define MPI_Send(arg...)  DTVS_Send(__FILE__, __LINE__, #arg, arg)
```

Each call to `MPI_Send()` is replaced with a call to `DTVS_Send()`. The macro argument `arg...` is a GNU extension to the ANSI C preprocessor that allows the remaining arguments to be used as a single entity. We prepend the name of the input file, the line number of the MPI function call, and a quoted version of the original argument list to the argument

list. For example, suppose that the following MPI function call appeared on line 27 of the file "`example.c`" whose execution is to be traced. After preprocessing, the resulting code would be the DTVS function call also shown below. Notice that the original argument list is quoted before any other substitutions are made. This allows the tracing system is accurately reproduce the call without consulting the source file.

```
MPI_Send(buffer, 2, MPI_INT, node, type, MPI_COMM_WORLD);

DTVS_Send("example.c", 27, "buffer, 2, MPI_INT, node, type, MPI_COMM_WORLD",
  buffer, 2, ((MPI_Datatype)6), node, type, 91);
```

Macros provide the DTVS functions all the information needed to trace the execution of the program without requiring the software engineer to alter the program. Initial and terminal events as well as communication events are traced when the header is included. The additional functionalities listed below are available at the cost of changes to the source code.

- Variables can be traced by identifying them with the `DTVS_trace_variable` macro. This must be done before the `MPI_Init()` function is called.
- Statements whose execution will create significant events can be identified using the `DTVS_Local` macro.
- Locations where values of variables should be recorded can be identified using the `DTVS_checkpoint()` function.

As a demonstration of the alterations needed to use the tracing facilities, an example MPI program is given in figure 5.1. The program creates several processes, the number

being dependent on command line arguments to the MPI initialization script. Each process enters a loop where it exchanges messages with each of the other processes. Repetition of the message exchange loop is controlled by a constant, **NUMCYCLES**. Communication is asynchronous using only the **MPI_Send** and **MPI_Recv** functions. The result of preprocessing is shown in figure 5.2. For brevity, only the **main** function is shown.

```
#include "mpi.h"
#include "DTVS_trace.h"

#define NUMCYCLES 1

void main(argc,argv)
int argc;
char *argv[ ];
{
  int me, nproc, cycle, type = 1, buffer[2], node;
  MPI_Status status;

  DTVS_trace_variable(me);
  DTVS_trace_variable(node);
  DTVS_trace_variable(buffer[0]);
  DTVS_trace_variable(buffer[1]);

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  for (cycle = 0; cycle<NUMCYCLES; cycle++)
    for (node = 0; node<nproc; node++)
      if (node != me) {
        DTVS_Local(buffer[0] = me);
        DTVS_Local(buffer[1] = node);
        MPI_Send(buffer, 2, MPI_INT, node, type, MPI_COMM_WORLD);
        MPI_Recv(buffer, 2, MPI_INT, node, type, MPI_COMM_WORLD, &status);
      }

  MPI_Finalize();
}
```

Figure 5.1: Example MPI program

```
void main(argc,argv)
int argc;
char *argv[ ];
{
  int me, nproc, cycle, type = 1, buffer[2], node;
  MPI_Status status;

  DTVS_trace_var("me", &(me));
  DTVS_trace_var("node", &(node));
  DTVS_trace_var("buffer[0]", &(buffer[0]));
  DTVS_trace_var("buffer[1]", &(buffer[1]));

  DTVS_Init("example.c", 18, "&argc, &argv", &argc, &argv);
  MPI_Comm_rank(91, &me);
  MPI_Comm_size(91, &nproc);

  for (cycle = 0; cycle< 1; cycle++)
    for (node = 0; node<nproc; node++)
      if (node != me) {
        buffer[0] = me; DTVS_Local("example.c", 25, "buffer[0] = me" );
        buffer[1] = node; DTVS_Local("example.c", 26, "buffer[1] = node" );
        DTVS_Send("example.c", 27,
          "buffer, 2, MPI_INT, node, type, MPI_COMM_WORLD",
          buffer, 2, ((MPI_Datatype)6), node, type, 91);
        DTVS_Recv("example.c", 28,
          "buffer, 2, MPI_INT, node, type, MPI_COMM_WORLD, &status",
          buffer, 2, ((MPI_Datatype)6), node, type, 91, &status);
      }

  DTVS_Finalize("example.c", 31);
}
```

Figure 5.2: Example MPI program after preprocessing

5.2 Trace Files

As the program is executed, three files are created for each process. The main file is named p*xx*.out where *xx* is a two digit process identifier. In MPI terms, the identifier is

the *rank* of the process where P_i has rank i. The other two files are p*xx*.out.Dppv and p*xx*.out.Dppe and contain variable values, indexes and offsets needed by the predicate evaluation subsystem. All trace files are stored in ASCII format for simplicity.

The main trace file is divided into two sections, variable declaration followed by execution trace. As shown in figure 5.3, the variable declaration section consists of a single line that lists that names of the variables in the order they were traced. Records representing event execution begin on the second line of the main trace file. Trace records represent the occurrence of a single event v and are of variable length. Each record is stored on a single line and begins with a one character indication of the record type followed by a variable name and a value. Since MPI uses buffers to send and receive messages, the value of the variable is always zero. This is not meant to imply that zeroes are sent and received, but that the information was not traced. The rationale is clarified when we consider the recorded value of an array of complex structures.

Each traced event v is considered significant and has a unique vector time associated with it. This vector time is the fourth item in the trace record. If the event is either a message transmission or receipt, a communication indicator is put in the record to quickly identify the direction of the message and the communicating partner event. For example, the string "R(P1.4)" would indicate that this event is the receipt of a message sent from process P_1 at local time 4. The event number included in a send event is the number of the send and not the event number of the receipt.

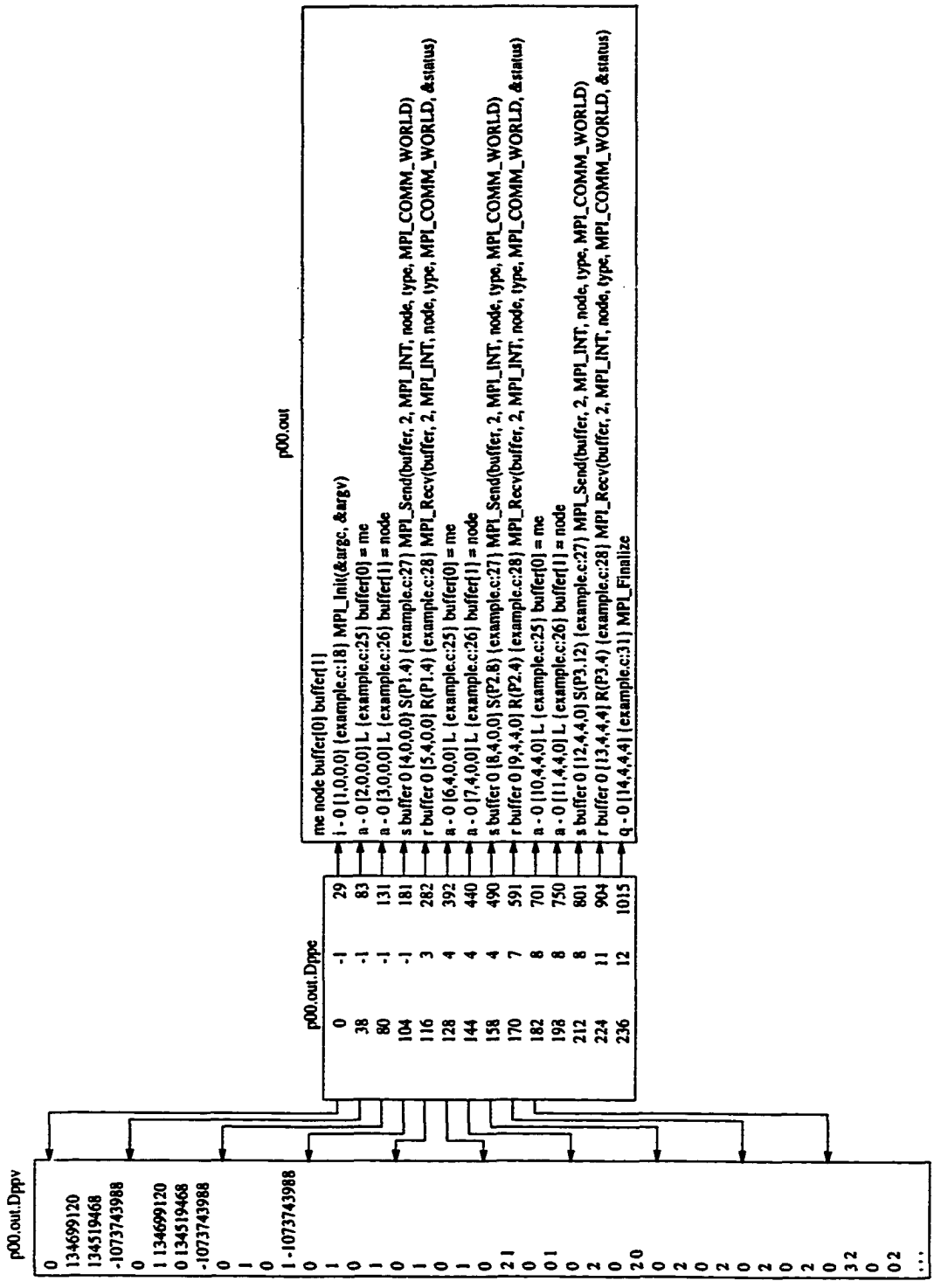


Figure 5.3: Layout of DTVS trace files

Next is the source code file name and the line number of the statement causing the event. A free form textual comment completes the record. In the current implementation of the tracing facilities, the comment is the source code line that caused the event. The file name and line number make this repetitious if the source file is available. The record format is given below and each item is briefly identified.

rec_type var val τ *commo* {*file:line*} *comment*

rec_type A single character indicating the type of event being traced. Currently limited to `a` (a local computation), `i` (initialization), `s` (message send), `r` (message receipt), `q` (termination), and `l` (label for next event).

var The name of the variable sent or received. If the event is not a communication event, the special value "-" indicates the absence of a variable.

val The value of the variable sent or received. Always zero in the current implementation.

τ The vector time associated with event v in the form $[\tau(v)[0], \tau(v)[1], \ldots, \tau(v)[n]]$.

commo If present, either `S(P`x`.`e`)` or `R(P`x`.`e`)` where x is the process ID of the transmitting process and e is the local clock value of the send event. A value of `L` is also possible if the event is a local computation event. This is only to maintain compatibility with a previous version of DTVS.

file The name of the source code file.

line The line number in *file* causing the event to occur.

comment A textual comment associated with the event. Currently the source statement causing the event.

Each time either an event occurs or `DTVS_checkpoint()` is executed, the values of all traced variables are recorded. When an event is written to the main trace file, the set of values for each variable is written to the variable value trace file (p*xx*`.out.Dppv`). Values are those attained since the previous significant event and are listed on a single line with the most recently acquired value listed first. Variable order is the same as in the variable declaration section of the main trace file. As with the main trace file, the variable value file is composed of variable length records.

The event index file (p*xx*`.out.Dppe`) is a fixed format file with a single record corresponding to each event recorded in the main trace file. Each record contains three, nine digit numbers separated by spaces and followed by a return character. Record i in the event index file begins with the byte offset in the variable value file where the values for event v_i begin. The next value in the index file is the record number of the last preceding communication event. The last value in the index file record is a byte offset into the main trace file where the record for event v_i begins.

Consider record number 5 (starting at zero) from the event index file of figure 5.3. The record contains the following values and matches event v where $\tau_0(v) = 6$. (Remember that the local component of the vector clock is initialized to 1.)

128 4 392

To locate the event record in the main index file, we read a line beginning at byte offset 392, the third value. We can locate the variable values associated with that event by consulting the first value in the record. They are found at offset 128 of the variable value file. To find the latest preceding communication event, we use the second value, 4. By multiplying this

value by the size of a record in the communication index file (thirty bytes) we compute the offset of the needed index record.

5.3 Constructing The Display

The main trace files for all processes are read at once. As events are processed, they are assigned a *height* that indicates the distance from the bottom of the time-space diagram that they should appear. For example, the initial event from each process should have a height of 1 and the next event, if not dependent on an event from another process, should have a height of 2. The display is not drawn at this point, only the lists are created. A list of messages is also constructed during file input. Algorithm 5.1 is used to construct the lists of both events and messages.

One event from each main trace file is read. These events are *current*. A height counter for each process is initialized to 1 and a loop is begun that determines which of the current events are *ready* to be added to the display. The decision is based on the vector times of the *current* events. The vector time of each event is compared to the vector times of each of the other events to determine a causal relationship. If it is determined that all causally preceding events were processed on a previous iteration of the loop, that is, $v' \not\rightarrow v$ for all other v' that are *current*, then v is *ready* to be displayed. In other words, the height of an event is one greater than the maximum height of any causally preceding event.

Each time a *ready* event is found it is assigned the value of the height counter. Another event is then read from the trace file but is not immediately made *current*. Instead, all newly read events are made current and the height counter is incremented before the next iteration

begins. The result is an assignment of a relative display coordinate, *(process,height)*, to each event.

```
Message_List = { }
for i = 0 to N do
    current_i = read_event() from P_i
    Event_List_i = { }
od
current_height = 1
while more events to process do
    /* find ready events */
    for i = 0 to N do
        ready_i = true
        for j = 0 to N, j ≠ i do
            if current_i → current_j
                ready_i = false
            fi
        od
    od

    /* set ready events and messages */
    for i = 0 to N do
        kif ready_i
            Event_List_i = Event_List_i ∪ {current_i, current_height}
            if current_i.rec_type = 'r'
                commo = current_i.commo
                h = height_of(commo)
                Message_List = Message_List ∪ {(commo.x,h),(i,current_height)}
            fi
            current_i = read_event() from P_i
        fi
    od
od
```

Algorithm 5.1: Construction of event and message display

Construction of the list of messages is also done during the input of the main trace files. The decision was made not to display messages that were never received. In other words, lost messages do not appear on the display. The send event is displayed without a message

arc leaving the node. Messages that are received are given two relative display coordinates. The first coordinate is the *(process,height)* of the event that sent the message and the second coordinate is the *(process,height)* of the event that received the message. The order here is important to display the arrowhead at the proper end of the message line.

When a message receipt event is found to be ready and assigned a height, the message coordinates are computed. By design, the trace record contains an easily accessible indication of the transmitting process ID and event number. We require that the string `R(P`x`.`e`)` where x is the process ID of the transmitting process and e is the event index in that process, follow the vector time. This may seem repetitious since we are also maintaining vector times for each event. However, a message may not be correctly identified by a receive event's vector time if receipt order is not identical to transmission order.

We know that the send event must have previously been assigned a display coordinate since it causally precedes the receipt. Otherwise the receipt would not be *ready*. We use the event number to retrieve the display height, h, of the sending event. We can now construct the display coordinates of the message as *[(x,h),(process,height)]*. These coordinates are added to the list of messages.

The maximum height of events and the number of processes are used to draw the time lines. Events are drawn using the display coordinates from the list. A directed line is added to the display to represent the message. Figure 5.4 shows the display constructed from the trace files built from the execution of the example MPI program with four processes.

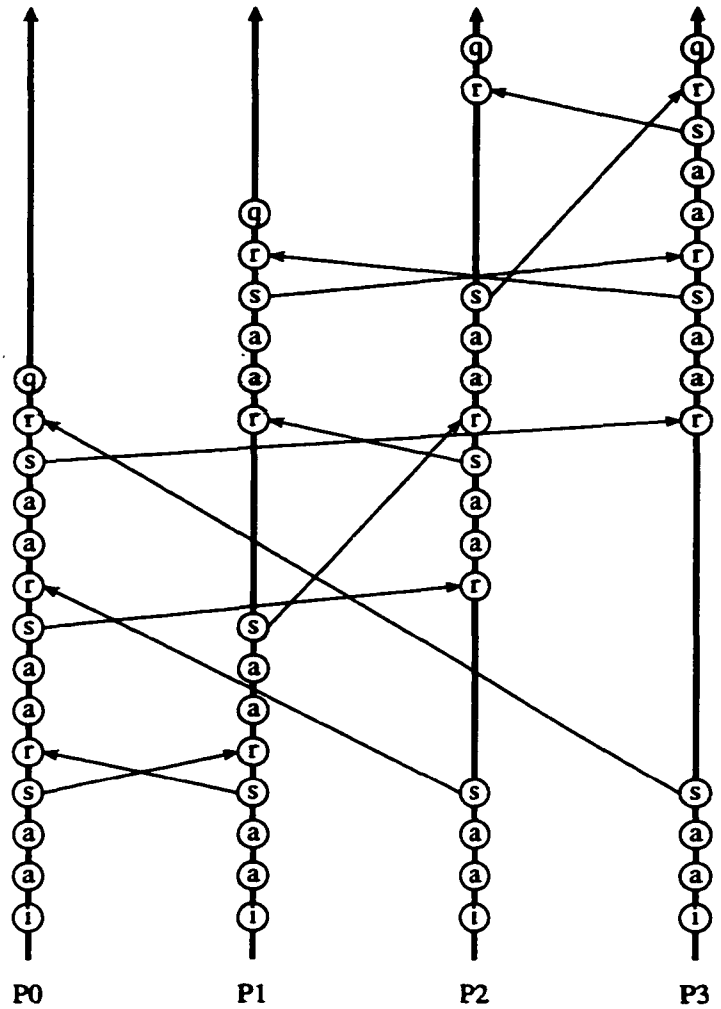


Figure 5.4: DTVS display of four process execution

Two visual clues are provided to indicate the relevance of an event to the software engineer. First, within the circular representation of an event is the event type. This is the single character type extracted from the main trace file. Second, a label can be inserted into the display using the `DTVS_Label()` function. This function takes two string parameters that are applied to the next event to occur in that process. The first parameter is displayed just to the right of the event oval and is usually a one word indicator of the specific event

type. The second parameter is displayed only when requested, thereby preventing cluttering of the display.

5.4 Usage of DTVS

Once the program has been written, compiled and executed, the trace files are ready to be examined. The DTVS program is executed with the names of the main trace files as arguments. The display is constructed as described above and presented to the user. Several options are now available. Using the left mouse button, events, labels and messages can be examined. The right mouse button selects and event as an anchor and highlights the events of other processes that are concurrent to the new anchor. Predicates can be defined and evaluated as if they had been placed in the source code directly after the anchor event.

5.4.1 Information Windows

Although the events are tagged with a type indicator, the user is commonly interested in more detailed information. We provide an event information window that is accessed by clicking on the event in question with the left mouse button. As shown in figure 5.5, the additional information about the event begins with the type of event being traced. In the figure, we have selected event eight of process P_1 which is a send event. The event type explicitly states that the destination of the message is P_2.

Also included in the event information window is the variable being sent and its value. In the example, the variable `buffer` was transmitted. Since the variable is an array, its value is not displayed and we instead place a `0` in the value field. Events that are neither

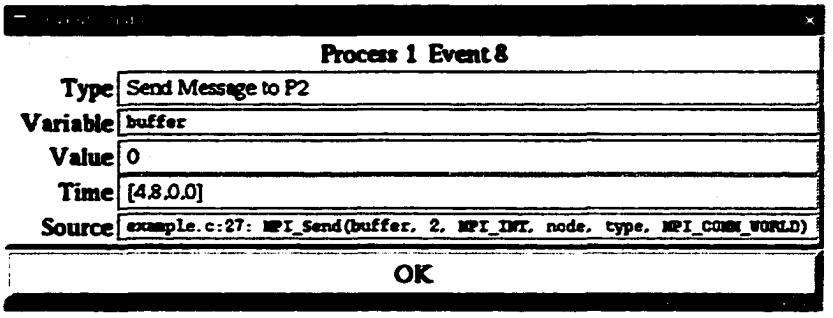


Figure 5.5: DTVS display of event information window

sends nor receives do not contain a variable and therefore have a dash in that entry of the information window.

The vector time associated with the event is presented as an ordered list of integers delimited by square brackets. The event in the figure has vector time `[4,8,0,0]`. The title of the window, `event Window 1-8`, indicates that the information is for event 8 of process P_1. We can verify this by examining component $\tau[1]$.

The final piece of information included in the event information window indicates the source code that generated the event. This line, given next to the label *Source*, contains three parts: the source file name, the source file line number, and a textual reproduction of the source statement. Again referring to the figure, the message transmission was caused by the execution of line number 27 of the file `example.c`. The source statement was as follows.

```
MPI_Send(buffer, 2, MPI_INT, node, type, MP_COMM_WORLD)
```

In a similar manner, messages and labels can be examined. Clicking on the directed arc depicting a message will cause the displays for the sending and receiving events to be

displayed. If the arc emanating from event 8 of process P_1 was selected, we would be presented with the event information windows for both event 8 of process P_1 and event 9 of process P_2. These are the send and receive events for that message.

Clicking the text of the label will display the full comment associated with that label. For example, assume that the following statement was inserted into the source code.

```
DTVS_Label("loop", "Beginning convergence loop with value 3.772")
```

When selected by clicking the word `loop`, a window would appear to display the remaining information, "`Beginning convergence loop with value 3.772.`" This facility provides the software engineer with the ability to easily annotate the display.

5.4.2 Concurrent Regions

The middle mouse button is used to select a particular event as an *anchor* event. An anchor event is needed both to display a concurrent region and to provide an assertion point for the predicate. As shown in figure 5.6, the selected event is filled in dark grey. A lighter grey is used to indicate the regions of concurrency. The figure shows that the final ten events of P_0 are concurrent to the anchor, a local computation in P_1.

Computing the events of each process that are within the concurrent region makes heavy use of event vector times. Suppose we wish to compute the region of events in P_0 that are concurrent to an event v in P_1 with vector time $[4, 6, 0, 0]$. The definition of vector time tells us that the last event in P_0 that causally precedes v is event number $\tau(v)_0 - 1$, or 3. Therefore, the first event that is possibly concurrent to v in P_0 is event number 4 if we assume that asynchronous communication was used. If the transmission was synchronous

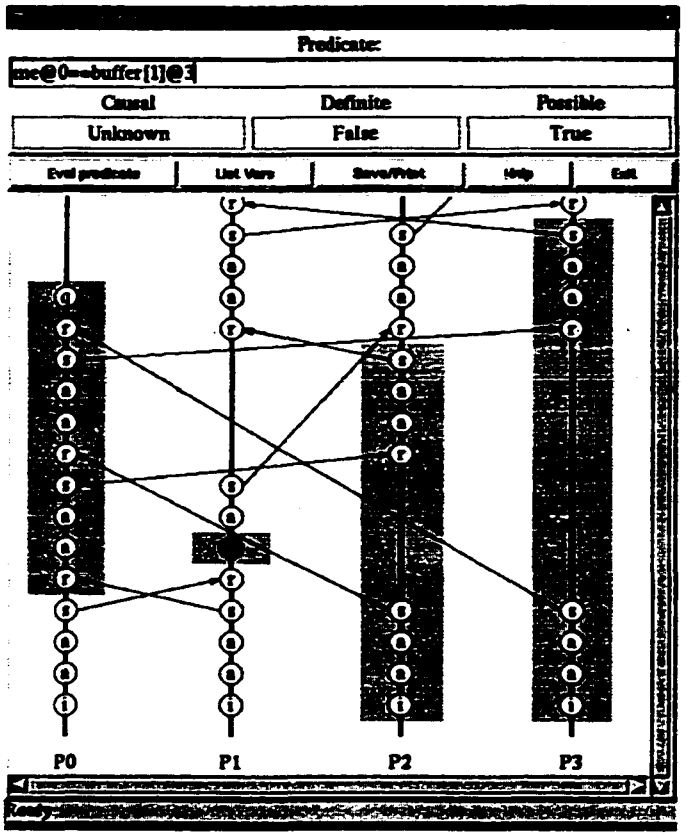


Figure 5.6: Snapshot of the DTVS display.

between v and event number 3 in P_0, then the first (and only) event concurrent to v in P_0 is event number 3.

Beginning with the first concurrent event, we sequentially search for the first event in P_0 that causally follows v. In this case, all events numbered greater than 3 are included in the concurrent region. If an event was found that causally follows v, then the preceding event in that process would be the last event concurrent to v. Algorithm 5.2 shows how the region of events in P_i that are concurrent to v is computed. The variables `first` and `last` indicate the bounds of concurrency in P_i.

```
first = τ(v)_i − 1
event = read_event(first)
if event → v
    first ++
fi
last = first - 1
event = read_event(last)
 /* Search comunication events until we find a causally succeeding event */
while v ↛ event do
    last ++
    if last ≥ Number_Events(P_i)
        last = Number_Events(P_i)
        break
    fi
    event = read_event(last)
od
 /* We found first succeeding, previous is last concurrent */
last = last - 1
if last < first
    return {}
else
    return {first, last}
fi
```

Algorithm 5.2: Computing events in P_i concurrent to v

When the loop terminates, we compare the values of `first` and `last`. If `last` is less than `first` then there are no events in P_i that are concurrent to v. Any other circumstance will indicate a concurrent region which includes events numbered `first` to `last`. Events in the concurrent region are highlighted in light grey. Events below the light grey region happen before v and events above the light grey region happen after v.

5.4.3 Predicate Evaluation

It is useful to be able to declare a global predicate and assert it in a program. In a conventional environment, the predicate is evaluated while the program is executing and the result is presented to the user. DTVS allows predicates to be defined after execution and asserted immediately after the anchor event. The assertion is then evaluated using three common methods and each result is presented to the user.

Definition of a predicate must adhere to certain rules. First, all variables must have been identified with the `DTVS_trace_variable()` function. Should the user forget which variables were traced, a window is available via the *List Vars* button that will list the names of each variable traced from each process. Each variable name must be followed by the process from which the value will be taken. The variable name and the process number will be separated by an "at" symbol. For example, if the variable `buffer[1]` from P_3 was used in a predicate, it would be identified by the string `buffer[1]@3`.

Second, the predicate must conform to the syntactical requirements of the Unix `expr` command. Below are some of the more common operators that can be used.

relational	`| & < <= = == != >= >`
numeric	`+ - * / %`
string	`: match substr index length`

The predicate of figure 5.6, `me@0==buffer[1]@3`, tests the equality of the variable `me` in process P_0 against the variable `buffer[1]` in process P_3. The current implementation of DTVS only allows variables of simple types in the expression.

Once the predicate is defined and an assertion point selected, evaluation can begin. The first assertion evaluation method was introduced by Simmons[41] and is *causal* evaluation. A single value for each variable is used to compute a causal value for the assertion. These are the values the variables had in the latest causally preceding events. For example, the causal value of `me@0` would be the value of `me` when the first send event of process P_0 occurred.

From the vector time of the anchor event we quickly identify the latest causally preceding event in each of the other processes. Through the event index file we have two step indirect access to both the event record and the values of the variables at that event. If multiple values are present for a particular variable, their order becomes significant. The first value found is the last value attained before that event occurred and is therefore the value we require.

Two concurrent evaluation methods, *definite* and *possible* [11, 17] are also included. If the assertion evaluates to true regardless of the total order of events, then *definite* is true. This is indicative of a system that must have attained the values necessary to make the predicate true during the examined execution. There is no temporal ordering of events that would falsify the assertion. If, on the other hand, there is a single total order of events that could have made the assertion true, then the value of *possible* will be true. A false value for *possible* means that no total ordering of events could verify the assertion.

Notice that the result of the causal assertion in the figure is `Unknown`. When we attempt to find a causally preceding event in P_3 from which to obtain the value of `buffer[1]`, we fail. No event exists in process P_3 that causally precedes the anchor event so we can neither verify nor refute the assertion. Likewise, it may not be possible to evaluate an assertion using the other two methods. If we include a variable from process P_j and the concurrent

region in P_j is empty, then no values exist for the assertion. Again, we present the **Unknown** result.

Alternatively we could assume that a concurrent region exists but that no events within the region were traced. We could also assume that the values at the beginning of the concurrent region were constant during the execution of the concurrent region. With these assumptions, we know that the values of the variables at the end of the latest causally preceding event should be used to evaluate the assertion. These values are the ones used in causal evaluation. We do not replicate the result and dismiss this approach.

We begin concurrent evaluation by constructing a list of values and vector times for each variable of the predicate. The lists are shown below. Each entry contains a set of values and a vector time. For example, the penultimate entry for `buffer[1]` has the value set {1, 3} and the vector time `[12, 4, 4, 7]`. Each entry states that the listed values were attained by the variable while executing the statements following the previous event and preceding the event with the associated vector time.

me@0	{{0} [5,4,0,0]}, {{0} [6,4,0,0]}, {{0} [7,4,0,0]}, {{0} [8,4,0,0]}, {{0} [9,4,4,0]}, {{0} [10,4,4,0]}, {{0} [11,4,4,0]}, {{0} [12,4,4,0]}, {{0} [13,4,4,4]}, {{0} [14,4,4,4]}
buffer[1]@3	{{-1073742836} [0,0,0,1]}, {{-1073742836} [0,0,0,2]}, {{0, -1073742836} [0,0,0,3]}, {{0} [0,0,0,4]}, {{0} [12,4,4,5]}, {{3, 0} [12,4,4,6]}, {{1, 3} [12,4,4,7]}, {{1} [12,4,4,8]}

Once the lists have been constructed, we enter a loop that tests each combination of values from the constructed lists. A set of values is taken and their vector times are compared. If the vector times are not concurrent, then the values are not considered. For example, when the vector time `[6, 4, 0, 0]` and `[12, 4, 4, 6]` are compared we

determine that the relationship is causal. Therefore, the values associated with these vector times are not compared.

When all vector times are concurrent, as is the case with the first values for each variable, predicate evaluation is done. If the result is true, we make *possible* true and if the result is false, we make *definite* false. The loop continues until all possible combinations of values are tested. It is possible exit the evaluation loop early if *definite* has been falsified and *possible* has been verified. In this case, no further evaluation will alter the results. The results are then displayed in the main window. If the loop finds no matching of values that can be used to evaluate the expression, then the result is **Unknown**.

In the example, when `0` with vector time `[5,4,0,0]` is compared to `-1073742836` with vector time `[0,0,0,1]`, we know that *definite* is false. The next comparison between `0` with vector time `[6,4,0,0]` and `0` with vector time `[0,0,0,1]` tells us that *possible* is true. The loop is terminated at this point and the results displayed.

5.5 An Example

As an example of the usefulness of the visualization technique, we present a distributed mutual exclusion algorithm based on token passing. To enter its critical section, processor P_i must receive the token from its left neighbor, P_{i-1}. To release the critical section token, P_i sends the token to its right neighbor, P_{i+1}. In each of the following examples mutual exclusion was not violated during execution. The source code for this example is given in appendix B.

Our example follows the token passing rules and begins with process P_{N-1} sending the

token to P_0. Each process enters its critical section twice before exiting. Figure 5.7 shows the display constructed from the correct execution of a four process system. Notice that a message transmitting the token lies between each of the labeled critical sections. This ensures that no two processes enter their critical sections concurrently.

As shown in figure 5.8, the correct operation is no longer immediately obvious when additional messages are added to the system. In this case, we have traced the same system with non-token messages being exchanged. The post-critical section code sends a message to each of the other processes. Before entering the critical section, all but one randomly chosen message is accepted. The result is that causal relationships are formed that are not part of the underlying mutual exclusion algorithm. With close examination, it can still be determined that no critical section violation occurs.

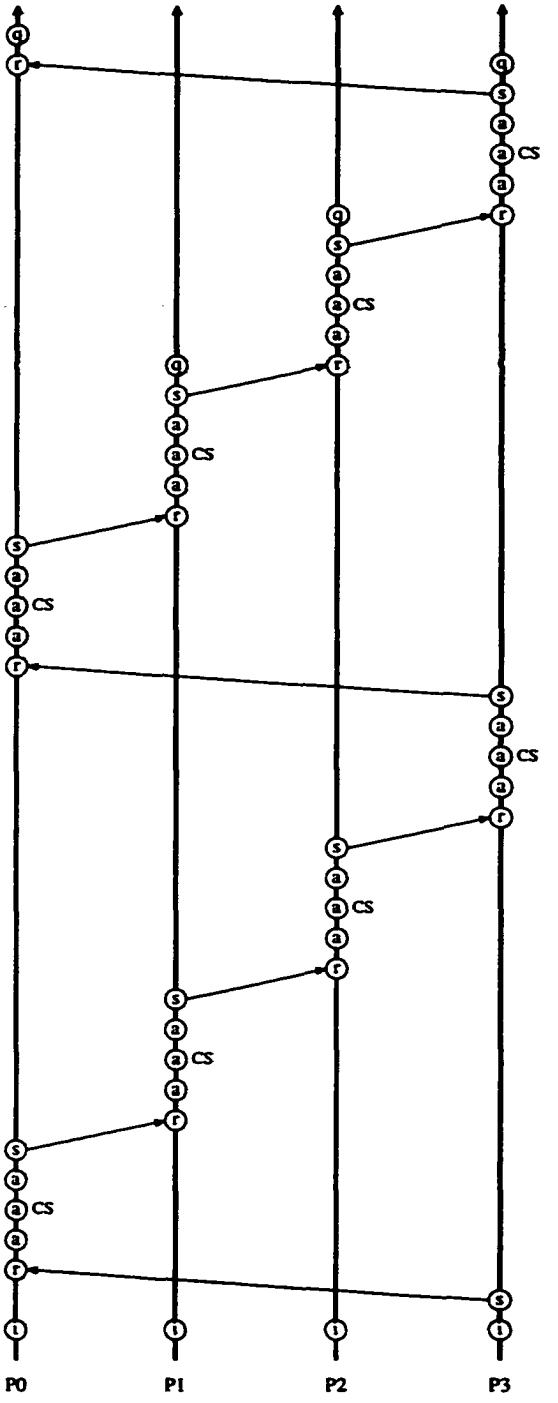


Figure 5.7: Correct execution of token ring mutual exclusion.

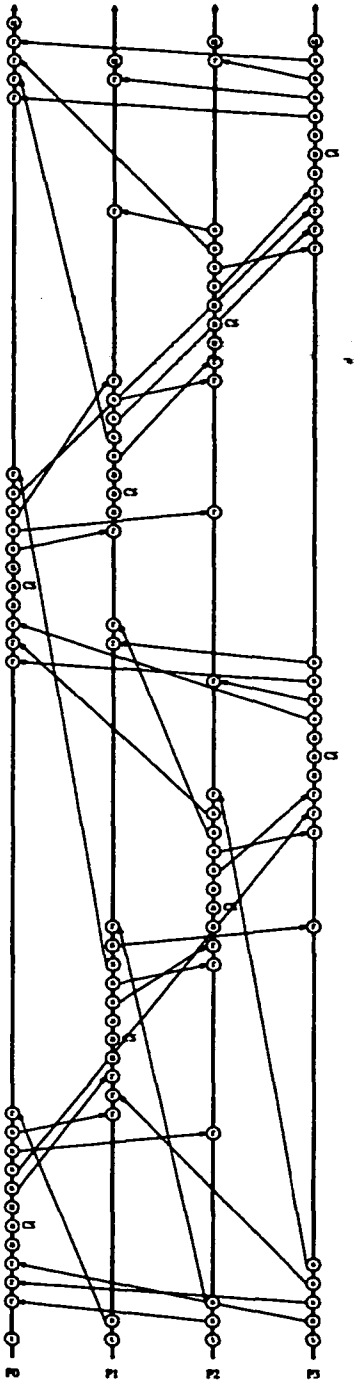


Figure 5.8: Correct execution of token ring mutual exclusion with additional messages.

In each of the correct examples, the receipt of the token message was accomplished with `MPI_Recv`, a blocking receive. That is, the function does not return until a message has arrived and is ready to be processed. To introduce an error into the program, we replace the function call with a textually similar function, `MPI_Irecv`. The new function is a nonblocking receive. It indicates to the underlying system that a message is expected and where it should be placed when it arrives and returns immediately without waiting for the arrival. The user can later check to see if the message actually has arrived. With nonblocking receives we incorrectly assume that the token has arrived when the function returns.

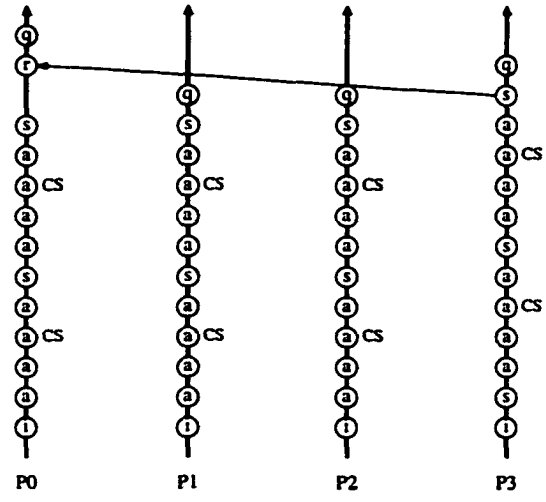


Figure 5.9: Incorrect execution of token ring mutual exclusion.

We begin by showing in figure 5.9 the display presented by DTVS without the additional, non-token messages. It is again clear what has happened. Since all critical sections are concurrent, we have obviously done something wrong. Selection of any event as an anchor would show that all events of the other processes are concurrent. The sole message in the

system is the token being sent from P_{N-1} back to P_0. The receipt uses the blocking version of the receive command and therefore actually receives the token.

If we overlay the additional messages back onto the execution, the problem is no longer obvious. Figure 5.10 shows the display constructed from the execution of the system with the non-token messages in place. It is interesting to note the near identical execution of the correct version of figure 5.8 and the faulty version of figure 5.10. If we examine the beginning of execution, we will notice that mutual exclusion is not violated. If we select a critical section event from any of the processes during the first round, we will see that no violation has occurred. This is due to the causality added by the non-token messages.

If we look at the top of the time-space diagram, we notice that the second entries into the critical sections of processes P_0 and P_1 seem to be concurrent. We can select either of the events as an anchor and the concurrent section will be highlighted. This will indeed show that the critical sections are concurrent. If the labels were not present, it would be difficult to identify the events that are within the critical section and those that are not. Concurrent regions may not be sufficient to notice the problem.

Suppose we select anchor event 14 from process P_2 as shown in figure 5.11. The labels, together with the concurrent regions, show that critical sections from processes P_0, P_1 and P_3 are all concurrent to message receipt. The question remains as to whether they are concurrent to each other. A simple predicate can check whether the value of `in_CS` from two processes is 1 at some concurrent instance.

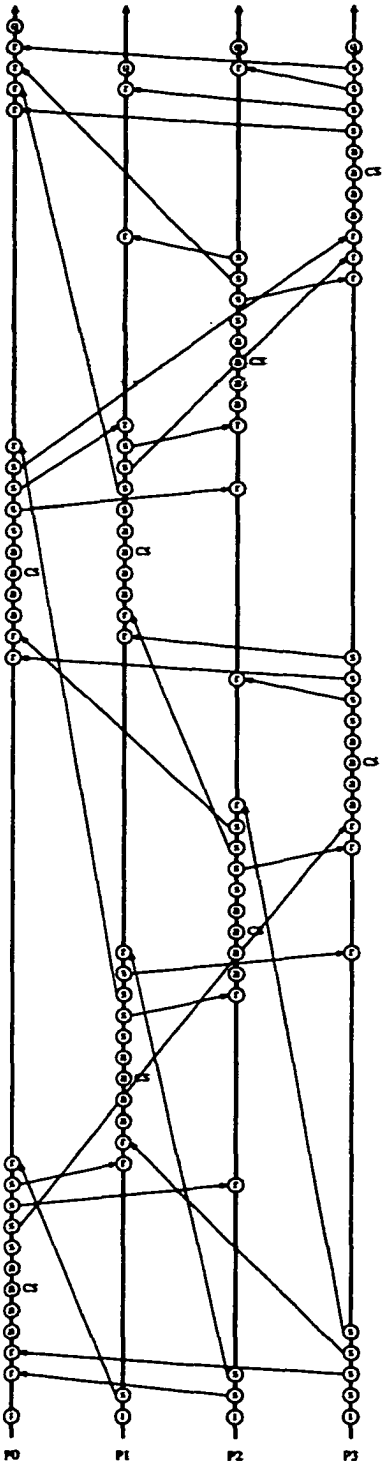


Figure 5.10: Incorrect execution of token ring mutual exclusion with additional messages.

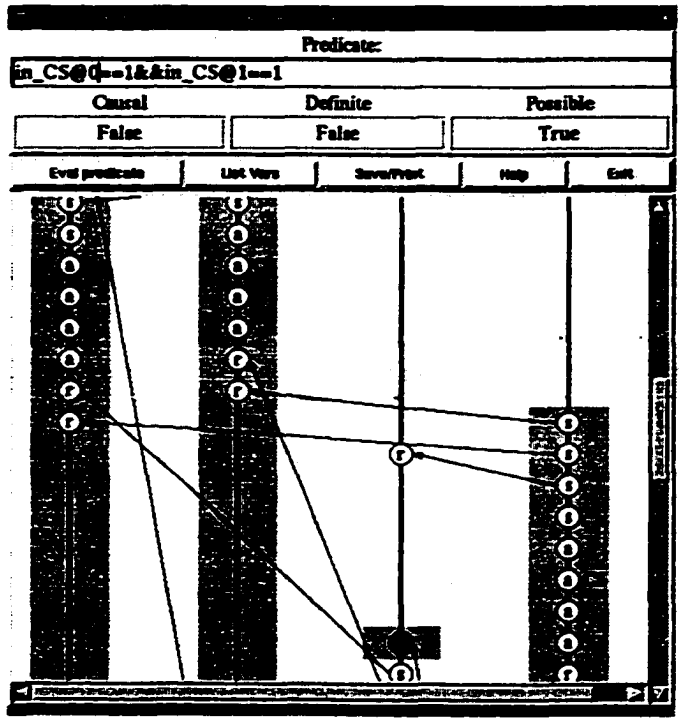


Figure 5.11: Assertion indicates mutual exclusion violation.

The predicate we entered was `in_CS@0==1&&in_CS@1==1`. This is the logical conjunction of the two critical section entry variables. We are not concerned in this example with the causal result. Instead we concentrate on the concurrent results. Since *definitely* is false, we know that some event orderings will not have P_0 and P_1 in their critical sections simultaneously. Since no runtime violation occurred, this must be the case. However, the *possibly* result of true tells us that even though mutual exclusion was not violated during execution, our algorithm does not guarantee it. In this case, mutual exclusion was a product of temporal ordering of events and not causal ordering. The next execution may cause the program to fail. The program may even function properly for months before the error becomes apparent with a failure at the worst possible moment.

Now that our assertion has told us of the existence of an error, we begin searching for its source. We first examine the events surrounding the critical section event. We begin by looking at the events immediately following the critical section of process P_0. The first event we encounter is an uninteresting local computation. Next we find a send event without an outgoing message arc. The event information window tells us that this is the transmission of the mutual exclusion token. The absence of the message arc indicates that the token was never received at the destination process. We now know what happened but have not determined the cause.

Working backwards from the critical section event of process P_1 we examine the events preceding the critical section code. We find a local computation that sets the value of `in_CS` to 1. The next event we examine is also tagged as a local computation event. The event information window however shows the function call `MPI_Irecv` as the source statement. It may not be obvious why the function is labeled a local computation. But consider its impact on the causal relationships of the system. Execution of a nonblocking receive simply indicates where a particular message should be stored. It does not wait for that message to arrive. Since no message has actually arrived, the causal link between sender and receiver has not been formed.

If, after the nonblocking receive has been executed, the `MPI_Wait` function is called, we insert a receive event into the trace file. The semantics this function are to wait until the indicated message arrives, store it as instructed by the nonblocking receive, and then return. If an `MPI_Wait` statement had been inserted immediately after the nonblocking receive, the resulting correct execution would be almost identical to figure 5.8.

Using the facilities provided by DTVS we have correctly identified the programming error. Furthermore, the error as not indicated by the failure of the program to perform correctly. DTVS is valuable in displaying not only the execution as it occurred but also in determining possible future execution scenarios.

Chapter 6

Conclusions

A program whose execution is intrinsically complex is commonly accepted as problematic in the sense that it does not fulfill its intended purpose. The complex interactions between constituent processes of a distributed system make a full understanding even more elusive. This research has focused on the accurate display of the system's execution so program developers can debug complex normally misunderstood executions. Specifically, we were interested in the accurate display of the concurrency between traced events.

This work was initiated by examining several existing techniques. While most were adequate for their intended purpose, none met all three of our criteria for a general purpose display: accurate, informative, and scalable. We closely examined the concurrency map, an incorrect solution, to determine the underlying reason for its fault. This solution, although not accurate, was both scalable and highly informative. During examination, a simple, four node graph was discovered that we labeled a quad-ring. This graph represents a common execution sequence of message transmission and receipt statements. We have rigorously proven that this graph, if present as a subgraph of the execution events, cannot be accurately displayed as a concurrency map. The quad-ring revealed that the concurrent

relationships we were attempting to display lacked the transitivity needed to be represented in two dimensions. This result lead our research to further investigate how many dimensions are required to display concurrent relationships, and if the number of dimensions needed depends on the number of processes.

We have developed a mathematical model that accurately characterizes both the concurrency and causality of the system. The model is theoretically scalable to any size system. Implementation of the model into a usable software engineering tool is problematic. No more than three processes can be simultaneously displayed. If more then a few events from each process are displayed, the presentation become too cluttered to distinguish individual events. While theoretically sound, the model has little practical value.

Our model furthered our understanding of the problem. We found that the unrestricted display of concurrency, a non transitive relation, cannot be accomplished in a single comprehensible manner. Restrictions of some type are necessary, but restricting the number of processes is not acceptable. As a result, we have developed a software prototype that displays all processes of the distributed system at once. The display is based on accepted graph models and presents the entire causal relation. The unavoidable but acceptable restriction is on the amount of concurrency presented at one time.

Our prototype allows the user of the system to select a single event and display the concurrency relation with respect to that event. Assuming that the user selects each event in turn, then all concurrency will eventually be displayed. We began by building tracing facilities on top of the widely used MPI message passing library and selecting the most commonly used communication paradigms. For each paradigm representative functions were selected from the myriad provided by MPI. Each function was augmented with tracing

facilities to create a nearly transparent interface between program execution and display.

The merit of our software is demonstrated by its accurate and informative depiction of the system. As an example, a simple distributed program is given that appears to operate correctly. The program, a mutual exclusion algorithm, does not fail to perform its intended task when tested. However, our display clearly shows that the correct operation is not due to a correct implementation but is instead attributed to a fortunate random ordering of concurrent events. It is possible that the program may operate correctly during testing and fail when put into operation.

We have demonstrated that additional features of our prototype allow the software engineer to quickly identify the cause of erroneous implementations. The software engineer can examine events and messages to determine whether the associated program statements are performing as expected. The ability to dynamically define predicates involving variables from the processes' executions has also been included. These predicates are asserted at locations in the program specified by the user and evaluated as if they had been present during execution. The combination of examination and assertion evaluation features provides an insightful and flexible tool for debugging complex distributed systems.

6.1 Future Research

Our developments have led us to several areas of future work. The first described areas are related to implementation improvements of the software prototype. The later described areas are theoretical in nature.

6.1.1 Software Enhancements

Our tracing facilities are currently a separate entity from the MPI libraries. To use our prototype the software engineer simply includes a header file. Macro definitions change selected MPI functions to the appropriate DTVS equivalents. This approach has several deficiencies.

We are required to create a DTVS function for each of the MPI functions we wish to trace. As the MPI group adds functions to their library, we must add the DTVS equivalent. A better, more permanent, solution would incorporate our tracing facilities into the base functions of MPI. For example, most message transmission functions eventually call a single routine, **send_message**, to perform the transfer. The same is true for message receipts. Most user callable receipt functions eventually use **recv_message** to accept an incoming message. Further research is needed to determine the consequences of incorporating our tracing facilities into these underlying functions.

Some user-callable MPI functions cannot be accurately traced without changing lower-level code. The synchronous send, for example, transmits a message and then waits for an acknowledgement of receipt. To implement vector time, we need to append the vector clock of the receiver onto the return acknowledgement message. This is not possible using the current approach since the actual send and receive are buried beneath several layers of software abstraction. We have instead used the standard **MPI_Send** and **MPI_Recv** functions in the DTVS wrapper to simulate the synchronous communication. Although it is functional and theoretically correct, it is neither an optimal nor elegant solution.

By introducing another level of software between the user program and the communication media, we increase the intrusiveness of the approach. It can be argued that adding one more layer to the ten already separating the user from the communication channel in the case of a basic send is not detrimental. However, we believe that every attempt should be made to avoid any runtime difference between a system being traced and a system not being traced. An advantage of incorporating the tracing facilities into the lowest level MPI functions would be to reduce their intrusiveness.

Another way of reducing the execution penalty of tracing would be to reduce the amount of data written to the trace files. This can be accomplished in several ways, some of which were mentioned in the discussion of the prototype. Instead of including the source code statement in the trace file, a file and line reference would be sufficient. This would require that the file is present during the display of the trace but since the tool is used during program creation, this should always be the case. It has been suggested that only the local component of the vector time be recorded in the trace files and other components be regenerated during display. More research is needed to verify this as a viable alternative. The trace files should also be stored in binary format.

We currently require the user to indicate the locations in the program where the values of traced variables should be recorded. Values are currently checked only when a significant event occurs or the user inserts a `DTVS_checkpoint` function call into the source code. It is possible that an important value of a variable is missed if checkpoints are not inserted in the proper places. We intend to explore the feasibility of automatically checking variable values without requiring the user to alter the source code. It is unclear whether an approach can be found that will be both transparent and an acceptable burden on execution.

Another requirement of the current implementation is that an anchor event must defined before a predicate can be evaluated. We believe that the ability to evaluate a predicate over a user-selectable region not associated with an anchor would be beneficial. For example, in the mutual exclusion program of chapter 5, we could have selected the entire program and evaluated the predicate over all concurrent cuts. The results may not be helpful unless some indication of the cut causing the evaluation results was present. A colored bar could indicate the events from each process that were used when the value of *possible* became true.

6.1.2 Theoretical

The ultimate goal of this research was to create a model that would allow the display of the entire concurrency relationship in a single picture. We have shown that the non-transitive nature of concurrency prevents the accurate display in a manner that clearly conveys the relationships between events. Instead, we have developed a technique that displays a subset of the concurrency relationship.

Each of the dimensions is used to show a particular aspect of the system's execution. One dimension indicates the physical separation of processes while another depicts the advance of logical time as events are executed. A two dimensional figure is sufficient to display the causal relationships between events. The third dimension is dedicated to the depiction of the concurrent relationships between events. Each plane in the third dimension shows the concurrent relationship with respect to a single anchor event. By selecting an event as an anchor, the user identifies the plane to be displayed. Once the anchor is selected, only the identified place is visible.

We plan to continue our exploration by attempting to increase the number of visible planes. Multiple planes may require the addition of color, texture or audible signals to retain the clarity of the original depiction. We will also consider the inclusion of a virtual reality display to allow the software engineer to be immersed in the execution trace.

Another possible approach is to reorient the displayed dimensions. It may be possible to use one of the primary planes to display the concurrent relationships. However, when increasing the information content of the picture, we must take care to retain a depiction from which valuable information can still be gleaned and avoid becoming another "angry fruit salad on drugs."[2]

Appendix A

Notation

The following is a list of notation used throughout the paper. Each symbol is followed by the page number of its introduction and a brief description of its meaning.

symbol	page	definition
N	15	The number of processes in the distributed system
P_i	15	Process i of the distributed system
v	16	An event executed in the distributed system
$P(v)$	17	The process in which event v is executed
v_i^k	17	The k^{th} event executed in process P_i
$v_i^\perp$	17	The initial event (preceding event 1) of process P_i
$v_i^\top$	17	The terminal event of process P_i
E	17	The set of events comprising the distributed system
E_i	17	The set of events executed by process P_i
S	17	The set of message transmission events

symbol	page	definition
R	17	The set of message receipt events
L	17	The set of local computation events
M	17	The set of communication events
m	17	A message in the distributed system
$S(m)$	17	The event causing the transmission of message m
$R(m)$	17	The event causing the receipt of message m
$\rightarrow$	19	The happens before or causal relation between events
/	20	A synchronous communication pair of events, a rendezvous
$\mathbf{H}$	20	The graph of the causal relationships between events
R_i^k	22	The causally equivalent region of events with respect to v_i^k
$\parallel$	23	The concurrent relation between events
$\bar{\mathbf{H}}$	24	The graph of the concurrent relationships between events
g(H)	24	The girth of graph G
CR_v	27	The concurrent region of event defined with respect to event v
τ	30	A vector representing causal dependencies
$\tau_i[j]$	30	The j^{th} component of the vector associated with process P_i
$\tau(m)$	30	The vector attached to message m
$\tau(v)$	30	The vector attached to event v
B_i^k	42	The k^{th} block of events in process P_i
$\mathcal{I}(B)$	43	The image or vertical location of block B in a concurrency map

symbol	page	definition
q^*	53	A quad_ring, or subgraph not displayable as a concurrency map
A	67	The events of E that succeed a selected event
B	67	The events of E that precede a selected event
C	67	The events of E that are concurrent to a selected event
α	72	The 1^{st} row of a concurrency map containing an event from A
β	72	The last row of a concurrency map containing an event from B
Ψ	72	A concurrency map
$C(v)^{\perp}$	118	A cut of the system execution that marks the beginning of CR_v
$C(v)^{\top}$	119	A cut of the system execution that marks the end of CR_v
$\mathcal{D}(v)$	121	The coordinates in N-space of event v
Δ_v	124	The distance from $\mathcal{D}(v)$ to the origin

Appendix B

Example MPI Program

```
/*****************************************************************/
/* Token_mutex.c                                                 */
/*                                                               */
/* Implementation of token passing Mutual exclusion              */
/* algorithm.                                                    */
/*                                                               */
/* If compiled with ERROR defined, a nonblocking receive         */
/* is used to obtain the CS entry token - an error.              */
/*                                                               */
/* If compiled with SIMPLE defined, all non-TOKEN messages       */
/* are removed for clarity.                                      */
/*                                                               */
/*****************************************************************/

#include <stdlib.h>
#include "mpi.h"
#include "DTVS_trace.h"

#define NUMCYCLES 2

#define TOKEN 111
#define MESSAGE 222

int me, nproc, in_CS = 0, half;
int *send_list, *recv_list;

/* called before entry into critical section */
void enter_CS()
{
```

```
  int buffer, left;
  MPI_Request request;
  MPI_Status status;

  left = (me + nproc - 1) % nproc;

#ifdef ERROR

  /* This is a NONblocking receive */
  MPI_Irecv (&buffer, 1, MPI_INT, left, TOKEN, MPI_COMM_WORLD, &request);

#else

  /* This is a blocking receive */
  MPI_Recv (&buffer, 1, MPI_INT, left, TOKEN, MPI_COMM_WORLD, &status);

#endif

  /* I now have the token */
  DTVS_Local(in_CS = 1);
}

/* Critical section code */
void do_CS_stuff()
{
  int garbage;

  DTVS_Label("CS","Complex calculations in critical section");
  DTVS_Local(garbage = 17);
}

/* Post critical section function */
void leave_CS()
{
  int buffer, right;

  DTVS_Local(in_CS = 0);

  right = (me + 1) % nproc;

  /* Pass the token to the right */
  MPI_Send(&buffer, 1, MPI_INT, right, TOKEN, MPI_COMM_WORLD);
}

/* randomize the process list */
```

```
void shuffle()
{
  int p, j, tmp;

  for(p=0; p<nproc; p++) {
    /* pick an element */
    j = random() % nproc;
    /* swap it with element p */
    tmp = send_list[j];
    send_list[j] = send_list[p];
    send_list[p] = tmp;

    /* pick an element */
    j = random() % nproc;
    /* swap it with element p */
    tmp = recv_list[j];
    recv_list[j] = recv_list[p];
    recv_list[p] = tmp;
  }
}

/* Do something before the critical section */
void pre_CS()
{
  int p, buffer;
  MPI_Status status;

  shuffle();

  /* Wait for all messages except 1                    */
  /* This should introduce lots of causality           */
  /* We miss one to allow mutex violation occassionally */
  for(p=1; p<nproc; p++)
    if (recv_list[p] != me)
      MPI_Recv(&buffer, 1, MPI_INT, recv_list[p], MESSAGE,
       MPI_COMM_WORLD, &status);
}

/* Do something after the critical section */
void post_CS()
{
  int p, buffer;
  MPI_Status status;

  /* now send messages to all other processes */
```

```
  for(p=0; p<nproc; p++)
    if (send_list[p] != me)
      MPI_Send(&buffer, 1, MPI_INT, send_list[p], MESSAGE, MPI_COMM_WORLD);

  /* receive message left over from pre_CS() receipts */
  if (recv_list[0] != me)
    MPI_Recv(&buffer, 1, MPI_INT, recv_list[0], MESSAGE,
     MPI_COMM_WORLD, &status);
}

/* initialize everything */
void initialize(int argc,char *argv[])
{
  int p, buffer;

  in_CS = 0;
  DTVS_trace_variable(in_CS);

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&me);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);

  /* initialize the lists of process numbers */
  send_list = (int *)malloc(sizeof(int) * nproc);
  recv_list = (int *)malloc(sizeof(int) * nproc);
  for(p=0; p<nproc; p++) {
    send_list[p] = p;
    recv_list[p] = p;
  }

  /* shuffle the process lists */
  shuffle();

  /* Last process sends token to 0 to get things started */
  if (me == nproc - 1)
    MPI_Send(&buffer, 1, MPI_INT, 0, TOKEN, MPI_COMM_WORLD);

#ifndef SIMPLE
  /* send wake-up messages to processes to the left */
  for(p=0; p<me; p++)
    MPI_Send(&buffer, 1, MPI_INT, p, MESSAGE, MPI_COMM_WORLD);
#endif

  /* seed the random number generator */
  srandom(48271 * (me + 1));
```

```
}

/* stop everything */
void finalize()
{
  int p, buffer;
  MPI_Status status;

  /* now pick the token */
  if (!me)
    MPI_Recv(&buffer, 1, MPI_INT, nproc-1, TOKEN, MPI_COMM_WORLD, &status);

#ifndef SIMPLE
  /* now pick the remaining messages */
  for(p=me+1; p<nproc; p++)
    MPI_Recv(&buffer, 1, MPI_INT, p, MESSAGE, MPI_COMM_WORLD, &status);
#endif

  sleep(1);

  MPI_Finalize();
}

void main(int argc,char *argv[])
{
  int cycle;

  initialize(argc, argv);

  /* the main processing loop */
  for (cycle = 0; cycle<NUMCYCLES; cycle++) {
#ifndef SIMPLE
    pre_CS();
#endif
    enter_CS();
    do_CS_stuff();
    leave_CS();
#ifndef SIMPLE
    post_CS();
#endif
  }

  finalize();
}
```

Bibliography

[1] M. AHUJA, A. D. KSHEMKALYANI, AND T. CARLSON. A basic unit of computation in distributed systems. *IEEE*, pages 12–19, 1990.

[2] B. P. MILLER AND C. MCDOWELL, EDS. Summary of ACM/ONR workshop on parallel and distributed debugging. *Operating Systems Review*, 27(4):8–23, 1993.

[3] P. C. BATES AND J. C. WILEDEN. High-level debugging of distributed systems: the behavioral abstraction approach. *Journal of Systems Software*, 3:255–264, 1983.

[4] ADAM BEGUELIN, JACK DONGARRA, AL GEIST, AND VAIDY SUNDERAM. Visualization and debugging in a heterogeneous environment. *Computer*, pages 89–95, June 1993.

[5] J. P. BLACK, M. H. COFFIN, D. J. TAYLOR, T. KUNZ, AND A. A. BASTEN. Linking specification, abstraction, and debugging. Technical Report TR-94-02, University of Waterloo, Canada, Department of Computer Science, November 1993.

[6] CHRIS CAERTS, RUDY LAUWEREINS, AND J. A. PEPERSTRAETE. PDG: A process-level debugger for concurrent programs in the grape parallel programming environment. Technical Report g95-01, Katholieke Universiteit Leuven, E.S.A.T. Laboratory, Heverlee, Belgium, 1994.

[7] T. A. CARGILL. The blit debugger. *Journal of Systems Software*, 3:277–284, 1983.

[8] BERNADETTE CHARRON-BOST. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39:11–16, July 1991.

[9] HERMAN CHERNOFF. The use of faces to represent points in k-dimensional space graphically. *Journal of the American Statistical Association*, 68(324):361–368, June 1973.

[10] W. H. CHEUNG. *Process and Event Abstractions for Debugging Distributed Programs.* PhD thesis, Univeristy of Waterloo, 1989.

[11] R. COOPER AND K. MARZULO. Consistent detection of global predicates. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 163–173, 1991.

[12] DENNIS EDWARDS AND PHIL KEARNS. DTVS: a distributed trace visualization system. In *Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing*, pages 281–288. IEEE Computer Society Press, 1994.

[13] M. G. FERNANDEZ AND S. GHOSH. Ddbx-LPP: A dynamic software tool for debugging asynchronous distributed algorithms on loosely coupled parallel processors. *Journal of Systems Software*, 22:27–43, 1993.

[14] C. J. FIDGE. Partial orders for parallel debugging. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 183–194, University of Queensland, May 1988.

[15] MESSAGE PASSING INTERFACE FORUM. MPI: A message-passing interface standard (version 1.1). Technical report, MPI-Forum, http://www.mpi-forum.org, 1995.

[16] MESSAGE PASSING INTERFACE FORUM. MPI-2: Extensions to the message-passing interface. Technical report, MPI-Forum, http://www.mpi-forum.org, July 1997.

[17] J FOWLER AND W. ZWAENEPOEL. Causal distributed breakpoints. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 134–141. IEEE Computer Society Press, 1990.

[18] PALLAS GMBH. Vampir 2.0 tutorial. Technical Report VA20-WTUT-11, Forschungszentrum Jülich, http://www.pallas.com, December 1998.

[19] G. GRÄTZER. *General Lattice Theory.* 75. Academic Press, Inc., The University of Manitoba, 1978.

[20] GEORGE GRÄTZER. *Lattice Theory: First Concepts and Distributive Lattices.* W. H. Freeman and Company, The University of Manitoba, 1971.

[21] FRANK HARARY. *Graph Theory.* Addison-Wesley Publishing Company, Inc., April 1971.

[22] DELBERT HART, EILEEN KRAEMER, AND GRUIA-CATALIN ROMAN. Interactive visual exploration of distributed computations. In *Proceedings of the Eleventh International Parallel Processing Symposium*, pages 121–127, Geneva, Switzerland, April 1997. IEEE.

[23] MICHAEL T. HEATH AND JENNIFER A. ETHERIDGE. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[24] MATTHEW S. HECHT. *Flow Analysis of Computer Programs.* 5. Elsevier Scientific Publishing Company, University of Maryland, 1977.

[25] DAVID P. HELMBOLD, CHARLES E. MCDOWELL, AND JIAN-ZHONG WANG. Traceviewer: A graphical browser for trace analysis. Technical report, University of California at Santa Cruz, October 1990.

[26] C.A.R. HOARE. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[27] SEKHAR SARUKKAI JERRY YAN AND PANKAJ MEHRA. Performance measurement, visualization and modeling of parallel and distributed porgrams using the aims toolkit. *Software - Practice and Experience*, 25(4):429–461, April 1995.

[28] R. KHANNA AND B. MCMILLIN. SMILI: Visualization of asynchronous massively parallel programs. *Journal of Systems Software*, 19:261–275, 1992.

[29] L. LAMPORT. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[30] ALLEN D. MALONY, DAVID H. HAMMERSLAG, AND DAVID J. JABLONOWSKI. Traceview: A trace visualization tool. *IEEE Software*, 1991.

[31] MasPar Computer Corporation, Sunnyvale, California. *MasPar Programming Environment (MPPE) - User Guide*, a5 edition, July 1992.

[32] FRIEDEMANN MATTERN. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, M. Cosnard et. al., editor, pages 215–226. Elsevier Science Publishers B. V., 1989.

[33] OLEG Y. NICKOLAYEV, PHILIP C. ROTH, AND DANIEL A. REED. Real-time statistical clustering for event trace reduction. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):144–159, Summer 1997.

[34] NORITAKA OSAWA. An enhanced 3-d animation tool for performance tuning of parallel programs based on dynamic models. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 72–80, Welches, OR, USA, August 1998. ACM.

[35] A. D. POLIMENI AND H. J. STRAIGHT. *Foundations of discrete mathematics*, chapter 4, pages 132–164. Brooks/Cole Publishing Company, Belmont, California, 1985.

[36] M. C. PONG. I-Pigs: An interactive graphical environment for concurrent programming. *The Computer Journal*, 34(4):320–330, 1991.

[37] D. A. REED, C. L. ELFORD, T. MADHYASTHA, W. H. SCULLIN, R. A. AYDT, AND E. SMIRNI. I/O, performance analysis, and performance data immersion. In *Proceedings of MASCOTS '96*, pages 1–12, San Jose, CA, USA, February 1996.

[38] G. C. ROMAN, K. C. COX, C. D.WILCOX, AND J. Y. PLUN. Pavane: A system for declarative visualization of concurrent computations. *Journal of Visual languages and Computing*, 3(1):161–193, 1992.

[39] REINHARD SCHWARZ AND FRIEDEMANN MATTERN. Detecting causal relationships in distributed computations: In search of the holy grail. Technical Report SFB 124-15/92, Department of Computer Science University of Kaiserslautern, 1992.

[40] SAMEER SHENDE, ALLEN D. MALONY, JANICE CUNY, AND KATHLEEN LINDLAN. Poratble profiling and tracing for parallel, scientific applications using c++. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 134–145, Welches, OR, USA, August 1998. ACM.

[41] SHARON SIMMONS AND PHIL KEARNS. A causal assert statement for distributed systems. In *Proceedings of the Seventh IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems*, M. H. Hamza, editor, pages 495–498. IASTED/ISMM, IASTED-ACTA Press, October 1995.

[42] JOHN T. STASKO. The PARADE environment for visualizing parallel program executions: A progress report. Technical Report GIT-GUV-95-03, Graphics, Visualization, and Usability Center, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, 1995.

[43] JOHN T. STASKO AND EILEEN KRAEMER. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, June 1993.

[44] A. M. STAVELY. Algorithms for analyzing concurrent software systems using derivatives. *Journal of Systems Software*, 11:3–20, 1990.

[45] J. M. STONE. A graphical representation of concurrent processes. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 226–235, 1988.

[46] GUIDO WIRTZ. The Meander language and programming environment. *Programming and Computer Software*, 21(1):9–16, 1995. Original Russian Text Copyright ©1995 by Programmirovanie, Wirtz.

[47] M. ZAKI, M. Y. EL-NAHAS, AND H. A. ALLAM. DPDP: An interactive debugger for parallel and distributed systems. *Journal of Systems Software*, 22:45–61, 1993.

VITA

Dennis Lee Edwards was born in Jackson Mississippi, January 20, 1966. He graduated from Richland Attendance Center in Richland Mississippi, May 1984. He then earned an A.S. in Data Processing from Hinds Community College in Raymond Mississippi, May 1986. At the University of Southern Mississippi in Hattiesburg Mississippi he earned both a B.S. and a M.S. in Computer Science, August 1988 and August 1991 respectively.

In August 1991, he entered the College of William and Mary as a graduate assistant in the Department of Computer Science. He earned his Ph.D., Decemeber 1999. He is currently an assistant professor in the Computing Department at the State University of West Georgia.